NPS ARCHIVE
2000.06
POLK, K.

DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY CA 93943-5101

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

TRANSPARENT DETECTION OF QOS VIOLATIONS FOR CONTINUOUS APPLICATIONS

by

Kendal V. Polk

June 2000

Thesis Advisor: Cynthia Irvine
Second Reader: Timothy Levin

Approved for public release; distribution is unlimited

DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE June 2000.	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE Transparent Detection of QoS Violations For Continuous Applications	5. FUNDING NUMBERS
6. AUTHOR(S) Polk, Kendal V.	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)* Resources Management Systems have the task of determining the structure, resource allocation, and scheduling of applications within their scope. One such system is the Management System for Heterogeneous Networks (MSHN) which uses its Client Library to gather knowledge of its environment. The Client Library is wrapped around each application to gather application status and resource usage information by intercepting and interpreting system calls. In previous work, the Client Library was utilized to provide status of an application at the end of the application's execution. This research focuses on a method to gather QoS information on continuous applications within mission-critical systems, while applications are running rather than after execution, without modification to the application's source code.

The Client Library has been modified to provide application execution information that is evaluated and compared against user-defined specifications. Any QoS violations result in a notification. This is an indicator for MSHNs scheduler to take corrective action such as adapting to use different resources or data formats.

When wrapped applications are used in conjunction with continuous monitoring, overhead is increased, which may be acceptable if transparent QoS monitoring is essential.

14. SUBJECT TERMS Quality of Service, Resource Management System, Desiderata, MSHN, Wrapper, QoS Violations, Client Library, Resource Monitoring	15. NUMBER OF PAGES 122
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18 298-102

Approved for public release; distribution is unlimited

TRANSPARENT DETECTION OF QOS VIOLATIONS FOR CONTINUOUS APPLICATIONS

Kendal V. Polk
Captain, United States Army
B.S., United States Military Academy, 1990

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 2000

TE SCHOOL

ABSTRACT

Resources Management Systems have the task of determining the structure, resource allocation, and scheduling of applications within their scope. One such system is the Management System for Heterogeneous Networks (MSHN) which uses its Client Library to gather knowledge of its environment. The Client Library is wrapped around each application to gather application status and resource usage information by intercepting and interpreting system calls. In previous work, the Client Library was utilized to provide status of an application at the end of the application's execution. This research focuses on a method to gather QoS information on continuous applications within mission-critical systems, while applications are running rather than after execution, without modification to the application's source code.

The Client Library has been modified to provide application execution information that is evaluated and compared against user-defined specifications. Any QoS violations result in a notification. This is an indicator for MSHNs scheduler to take corrective action such as adapting to use different resources or data formats.

When wrapped applications are used in conjunction with continuous monitoring, overhead is increased, which may be acceptable if transparent QoS monitoring is essential.

TABLE OF CONTENTS

I. INTRODUCTION 1

A. BACKGROUND 1

1. Quality of Service (QoS) 1

2. Commercial Off the Shelf (COTS) 3

3. Resource Management Systems (RMS) 5

B. MOTIVATION 6

C. SCOPE OF THE THESIS 7

D. ORGANIZATION 7

II. REAL-TIME, DISTRIBUTED APPLICATIONS 9

A. AEGIS 9

B. DESIDERATA 11

1. Overview 11

2. Dynamic Paths 11

3. System Architecture 13

C. QUALITY OF SERVICE REQUIREMENTS 17

1. Overview 17

2. Timeliness 17

3. Accuracy 18

4. Precision 18

D. SUMMARY 20

III. RESOURCE MANAGEMENT TOOLS 21

A. GLOBUS 22

B. ERDOS 24

C. MSHN 27

D. SUMMARY 30

IV. QOS VIOLATION MONITORING 31

A. DESIDERATA 32

B. MSHN 34

1. Transparent Path Monitoring System(TPMS) 35

2. MSHN Integration 39

C. SUMMARY 41

V. EXPERIMENTS AND RESULTS 43

A. EXPERIMENTAL DESIGN 43

1. Violation Detection 45

2. RMS Execution 45

B. RESULTS OF EXPERIMENTS 46

1. Violation Detection 46

2. *Effect of Monitoring on Path Latency* 47

VI. CONCLUSIONS AND FUTURE WORK **51**

A. CONCLUSION 51
B. FUTURE WORK 52
1. *Expansion of Existing MSHN Wrapper Functionality* 52
2. *Integration of Windows Application Monitoring* 53
3. *Intergration of TPMS into MSHN* 53

APPENDIX A. DATABASE DESIGN **55**

A. SPECIFICATION DATABASE 55
B. PATH DATABASE 56

APPENDIX B. TMPS MODULE SPECIFICATION **59**

A. SPECDB MODULE 59
B. PATHDB MODULE 60
1. *pathDB.h* 60
2. *pathDBInstance.h* 61
C. PATHTIMER MODULE 62
D. EVALUATE AND ALERT MODULE 62
E. CONTROL MODULE 62

APPENDIX C. TPMS SOURCE CODE **65**

A. SPECDB MODULE 65
B. PATHDB MODULE 74
1. *PathDB Source Code* 74
2. *PathDBInstance Source Code* 86
C. PATHTIMER MODULE 93
D. EVALUATE AND ALERT MODULE 94
E. CONTROL MODULE 96

LIST OF REFERENCES **99**

INITIAL DISTRIBUTION LIST **103**

LIST OF FIGURES

Figure 1. QoS Specification Hierarchy [CHAT98] 2
Figure 2. Example Resource Management System 5
Figure 3. AEGIS Architecture [PERR98] 10
Figure 4. Logical architecture management software [DESI98] 14
Figure 5. Desiderata software for resource and QoS management [DESI98] 15
Figure 6. Precision vs. Accuracy 19
Figure 7. The Globus resource management architecture, showing how RSL specifications pass between application, resource brokers, resource co-allocators, and local managers. [CZJA97] 24
Figure 8. ERDOS System Architecture [CHAT98] 26
Figure 9. ERDOS Middleware QoS Architecture [CHAT98] 27
Figure 10. MSHN Conceptual Architecture [HENS99] 29
Figure 11. Timestamp Sequencing [DESI98] 33
Figure 12: TPMS Representation in Regards to MSHN 36
Figure 13: PathDB Module Initialization 37
Figure 14. DynBench Application Communications [DESI98] 44
Figure 15. DynBench Overhead in Desiderata and TPMS 49
Figure 16. Specification Database in Memory 56

LIST OF TABLES

Table 1. Timestamp Sequencing 38
Table 2. Deadline Derivation 45
Table 3. DynBench Execution Data 48
Table 4. Specification Database 55
Table 5. Path Database 57
Table 6. Path Database Instance Definition 57

ACKNOWLEDGEMENT

First, I thank my supportive family for their understanding and patience. I then thank Dr. Cynthia Irvine and Tim Levin for their help, knowledge and expertise. I could not have done it without them. And finally I thank all the students I met here for their friendship and kind words.

I. INTRODUCTION

Determining whether distributed, real-time applications, such as those for Command and Control, need to be re-scheduled or re-configured for different resource allocations requires that a resource management system (RMS) quickly detects, or better yet predicts and reacts to predicted violations of Quality of Service (QoS) requirements. Current resource management systems require that the source code be modified to report such violations to the RMS. Unfortunately, this often precludes introducing Commercial Off the Shelf (COTS) or legacy software into such systems, even if the COTS or legacy algorithms are superior, because source code is either not available or would require a very expensive license. This thesis examines an approach for detecting QoS requirement violations in Resource Management Systems (RMS), which does not require source code access or modification.

A. BACKGROUND

1. Quality of Service (QoS)

QoS is a term used by many research groups in many different contexts. Before defining QoS, a definition of *service* is necessary. "**Service** is work done on behalf of someone, whom we denote as a client."[CHAT98] For example, an end-to-end communication application is a service executed for the end user. This service can be executed to varying degrees of quality. **Quality of Service** is the functionality that a client receives from a service executed at a given level of quality. The client receives a

certain level of functionality if the service provides a certain level of quality. Service specific, QoS parameters define this level of quality [CHAT98].

QoS parameters or specifications can be grouped into metrics or policies. Metrics specify QoS parameters that are quantifiable, whereas polices define system actions based upon system state. The following chart presents a sample QoS hierarchy:

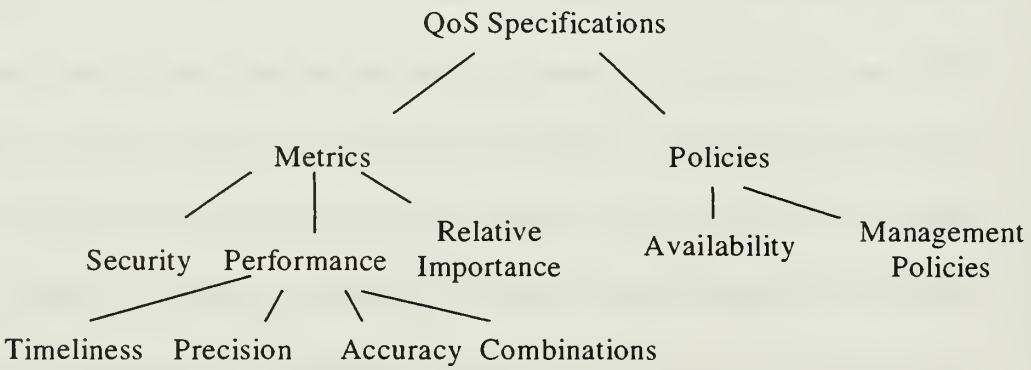

Figure 1. QoS Specification Hierarchy [CHAT98]

Normally when researchers discuss QoS they focus on performance metrics. Timeliness is generally a representation of the time required to execute a given piece of work. Example timeliness parameters are total time (begin to end), start time (earliest/latest) for a task, and deadline (earliest/latest) for a task to complete. Precision relates to data volume quantities, and can be represented as the precision of content for input and output data, and the representation for input and output data. Errors introduced into data by a service define accuracy. Accuracy parameters relate to the content for input and output data and to the representation for input and output data. [CHAT98][CLARK98]

2. Commercial Off the Shelf (COTS)

Government policies have recently undergone a change with respect to the acquisition of computer systems. There has been a shift in the DoD to evaluate the use of COTS products in a new system prior to its development [SLED98]. The rising costs of system development and shrinking budgets necessitate the interest in less costly COTS technology. "In systems where the use of existing commercial components is both possible and feasible, it is no longer acceptable for the government to specify, build and maintain a large array of comparable proprietary products." [SLED98]

Government procurement requires precision to ensure cost effectiveness and interoperability. COTS are referred to as "commercial items" under the Federal Acquisition Regulations (FARs) [FAR 96] and follow these general characteristics:

- exists *a priori,*
- available to the general public, and
- can be bought (or leased or licensed) [OBER97]

Those implementing systems using COTS components must realize that there are issues with system distribution, interface standards, and legacy system reengineering that affect a COTS-based distributed implementation. Supportability over the equipment or software's lifetime is also a key issue and must be analyzed on a case-by-case basis. "Each system or equipment must be scrutinized more closely than MIL-SPEC programs because of the dynamic commercial infrastructure, and technology turnover."[VIRT98]

There are seven major documents that guide the federal government, and especially the DoD, on the use of commercially available information technology products. Their focus on the use of commercial IT products are summarized as follows [OBER97]:

- Clinger-Cohen: "increase acquisition and incorporation of commercial technology"
- FAR: "acquire commercial and nondevelopmental items (C/NDI) when available to meet the needs [of the program]" (The FAR also requires primes and subcontractors at all tiers to incorporate C/NDI to the maximum extent practical.)
- DoD 5000.1: "If use or modification of existing...equipment will not meet the need, give top priority to...commercially available equipment."
- DoD 5000.2-R: "Consider C/NDI [to be] the primary source of supply"
- Joint Technical Architecture (JTA): "specifies a set of performance-based, primarily commercial, information processing, transfer, content, format and security standards.
- Defense Information Infrastructure (DII) Common Operating Environment (COE) : "consists of an approach for building interoperable systems; a collection of reusable software components; a software infrastructure for supporting mission area applications"

Each of these policy documents has the common thread that they mandate or encourage the use of commercial items, standards (non-governmental), technology, and/or best practices.[OBER97]

3. Resource Management Systems (RMS)

QoS is easy to provide to applications if these applications do not share resources. Broadcast and cable TV providers are able to guarantee a certain level of quality of service because each channel is assigned a certain frequency (i.e. no resources are shared)[CHAT98]. In a computing environment where resources are scarce, a degree of QoS is desired, and multiple services utilize these same resources, a resource management system is needed.

"**Resource Management** determines *what* service to perform and *where* and *when* to perform the service[CHAT98]." What determines the structure of the service; where determines which resources to allocate to each service; when determines the schedule of how to execute a set of services on a resource. A sample RMS is below:

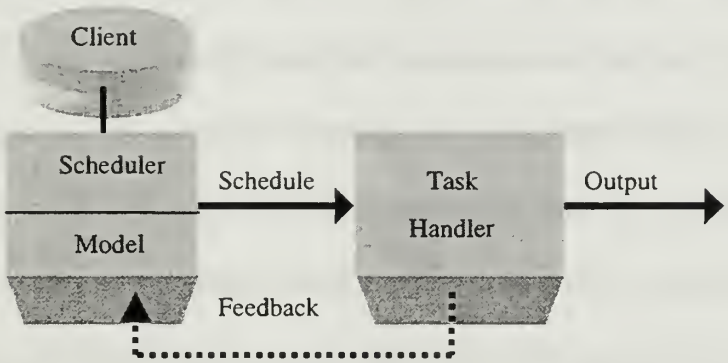

Figure 2. Example Resource Management System

In this case the client requests a service from the RMS scheduler and, based upon the model, which is a database that contains information about the resources necessary to execute the service (what and where), the scheduler determines when to execute this

service based upon current resource status and QoS parameters. Once execution has begun the task handler provides feedback to scheduler and the model to enact any necessary schedule changes. This is known as adaptive resource management.

B. MOTIVATION

The research performed in this thesis provides a basis for efficient QoS measurement in a distributed, heterogeneous computing environment. Specifically, this thesis research will focus on identifying the following: QoS requirements in this type of computing environment, the data that need to be captured/sent to a QoS measurement mechanism, and the use of this data to detect violations of these requirements by COTS applications. QoS violation detection techniques that can be applied to a broad range of software will allow more COTS applications to be introduced into future real-time, warfighting systems, perhaps enhancing speed and interoperability.

The DARPA-sponsored Management System for Heterogeneous Networks (MSHN) project, currently underway at NPS, is a subcomponent of the DARPA QUORUM program. The goal of this project is to provide Quality of Service (QoS) for both operational and planning applications that are able to use multiple sets of resources, while accounting for priorities and environments that change dynamically. The RMS for MSHN is equipped with algorithms to maximize overall benefit rather than benefit to a particular user. Thus an individual may not get everything he wants, i.e. "ideal", but may get "pretty good" instead. Currently, most systems implement QoS violation detection by directing that the monitored applications pass messages to the RMS. This method is not

applicable to COTS or legacy software that cannot be retrofitted to pass these types of messages. Techniques must be developed to detect and report QoS violations without the modification of source code. We call this transparent detection of QoS violation. The MSHN CL was developed to provide such transparent detection. However, CL only measures QoS at the end of a user's application. This has two problems. First, an RMS does not have the opportunity to adapt if QoS information is received after task completion. And second, this mechanism cannot be used with tasks that run continuously (e.g. sensor tasks).

C. SCOPE OF THE THESIS

This thesis research focuses on the transparent detection of QoS violations to enable timely system re-configuration (adaptation). Initially, QoS requirements for real-time, distributed applications will be identified and characterized based upon user criteria. Secondly, an experimental analysis of transparent QoS detection overhead and timeliness will be performed on the continuously executing DynBench applications from the Desiderata RMS. Conclusions generated from the analysis will give focus to future experiments and modifications of current techniques for QoS detection.

D. ORGANIZATION

Chapter II discusses the QoS requirements of a real-time distributed system with a focus on the applications Desiderata and AEGIS. Chapter III presents current Resource Management tools and a detailed overview of the MSHN project. Chapter IV describes the modification to MSHN's monitoring library that will enable dynamically adaptable

QoS violation monitoring. Chapter V discusses the experiments conducted to determine the timeliness and overhead of this approach to QoS detection. Conclusions from this thesis research and suggested future work that this research may lead to are described in Chapter VI.

II. REAL-TIME, DISTRIBUTED APPLICATIONS

This chapter gives an overview of real-time, distributed applications and their QoS requirements. Section A focuses on the AEGIS system, its components and functions. It will analyze the system progression and applicability to command and control situations. Section B focuses on the Desiderata project, its application to real-time distributed systems, and its use of dynamic paths. Section C presents and analyzes the QoS requirements of real-time distributed applications.

A. AEGIS

An early example of a real-time distributed application is the AEGIS Weapon System developed for the United States Navy in the 1970's. Its purpose is to defeat hostile anti-ship cruise missile technology. Its search and track space envelope covers over 100,000 square miles. It utilizes a network operating system that allows its four main systems to "talk" to one another to detect, track and engage targets. The systems are [PERR98]:

- Command and Decision Module-- Interface between man and machine.
- Weapon Control System-- Controls the processes that engage weapons.
- Radar System--Controls radar to search for and track/evaluate targets, missiles.
- Display System--- Supports Large Screen Display processing.

For a detailed description of all components see [PERI98]. The following is a graphical representation of how the systems interact:

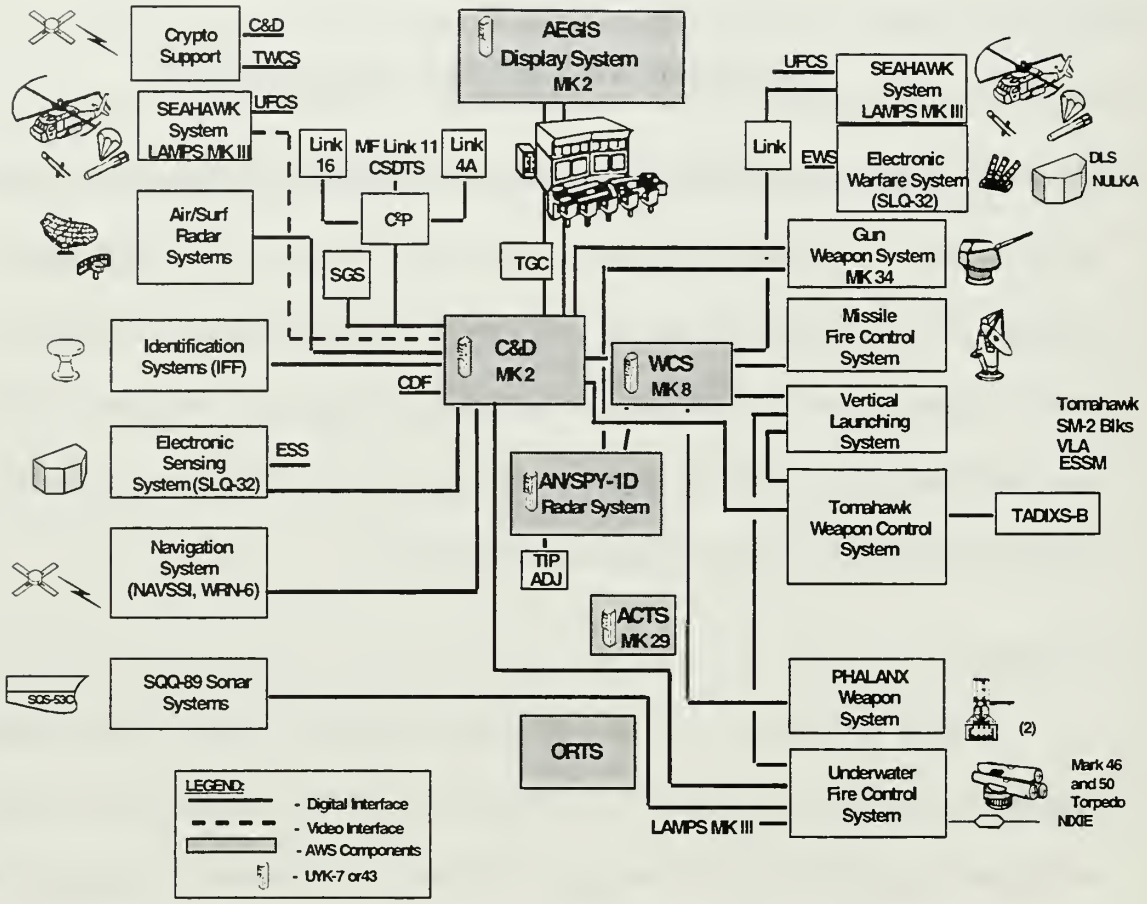

Figure 3. AEGIS Architecture [PERR98]

As this system approaches its third decade, developers are attempting to integrate the computing advances that have occurred since its inception. Faster hardware, networks, and targets have required developers to modify current scheduling algorithms and software to take advantage of real-time, heterogeneous computing environments. Determining and implementing these modifications are not problems specific to the U.S. Navy but to all of the DOD, which is why projects are being funded throughout the country to address them. One such project is Desiderata. [DESI98]

B. DESIDERATA

1. Overview

Desiderata is focused on the development of next-generation, distributed systems for combat. These types of systems have strict QoS objectives. They must contain some underlying mechanism enforcing a reliability policy, react to threats in a timely manner, and always be available in hostile environments. Resource usage must be efficient and scalability must be possible to "address the ever-increasing complexity of scenarios that confront such systems [DESI98]." To provide QoS, Desiderata focuses on the following:

- QoS Specification
- QoS metrics
- dynamic QoS management, and
- benchmarking

The name Desiderata is derived from its applicability to Dynamic, Scalable, Dependable, Real-Time systems. Desiderata is a testbed project for the US Navy to enhance QoS management technology in a shipboard distributed computing environment. This technology includes its own specification language and QoS management programs that support dynamic *path-based* systems. During application execution the QoS management system benchmarks application information which is used to facilitate resource management.

2. Dynamic Paths

A dynamic path is an entity used in Desiderata's QoS assessment to identify the start and end point of a series of connected actions. Each action is usually performed by a

separate application. These paths may have resource or timing constraints that are used to determine adherence to QoS specification. As with many air defense type systems, Desiderata begins with the *threat assessment path.* These actions include sensor (radar) input, filtering, filter management, and evaluation. The information passed along the path is considered dynamic because the sensor's input may range from a few to several thousand tracks and cannot be determined in advance. This type of path is classified as "continuous" because it is in constant operation [DESI98]. Normally there will be some timeliness requirement determined for the end-to-end latency of processing one set of sensor input.

The next path begins once the threat assessment path determines that there is a potential threat and an action against that threat is necessary. This type of path is called "transient" because it is invoked in response to sensor input. [DESI98] Time is also a critical factor in this path. The path latency objective is key in these types of systems since they may involve mission-critical or safety-critical operations. Desiderata refers to this as the *engagement path*,

The final path is the *missile guidance path*, which is "activated by an action initiation event and deactivated upon completion of the action. [DESI98]" Since this path behaves like a continuous path once it becomes active, it is called "quasi-continuous". Once the system fires a missile (activated), it will continually give guidance commands to that missile until it explodes (deactivated). The characteristics (speed, distance, altitude) of the threat cause the iteration time of a cycle to be dynamically configured.

3. System Architecture

Actions in the dynamic paths (real-time paths) send time-stamped messages to the *QoS metrics* component. This component then calculates whether the path-level QoS metrics are being met and sends this information to the *QoS diagnosis* component when a violation is detected. The diagnosing component advises the *action selection* component of the cause of the poor QoS and recommends actions such as moving to a different host or program replication to improve QoS. The *action selection* component determines the best of the recommended actions to execute. The *allocation analysis* component determines a suitable way to allocate resources for these actions by consulting the *resource discovery* for local area network (LAN) *hardware metrics*. This component then requests the *allocation enactment* component to execute these actions. At heart of this architecture are the *spec files* which contain path latency requirements, startup LAN and Host resource specifications (e.g. SPARC, Windows® NT, 200Mhz), resource locations, and real-time path designations. See Figure 4.

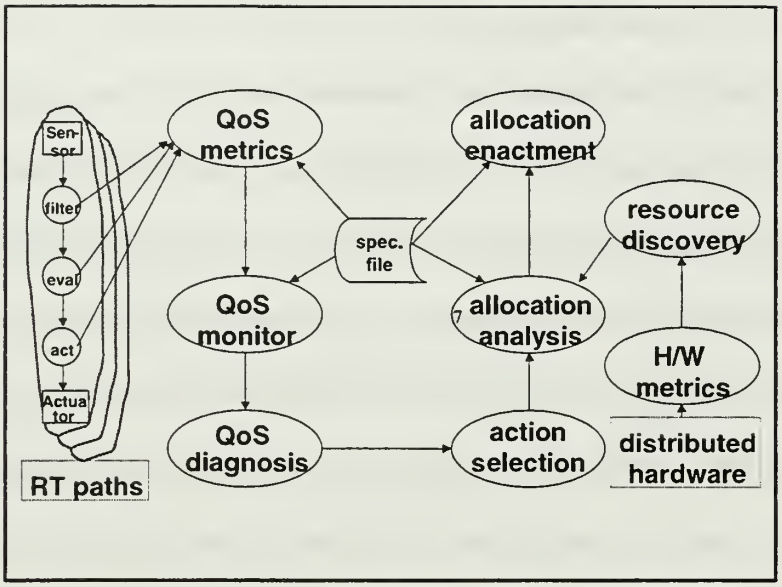

Figure 4. Logical architecture management software [DESI98]

Desiderata's system architecture is displayed as follows in Figure 5:

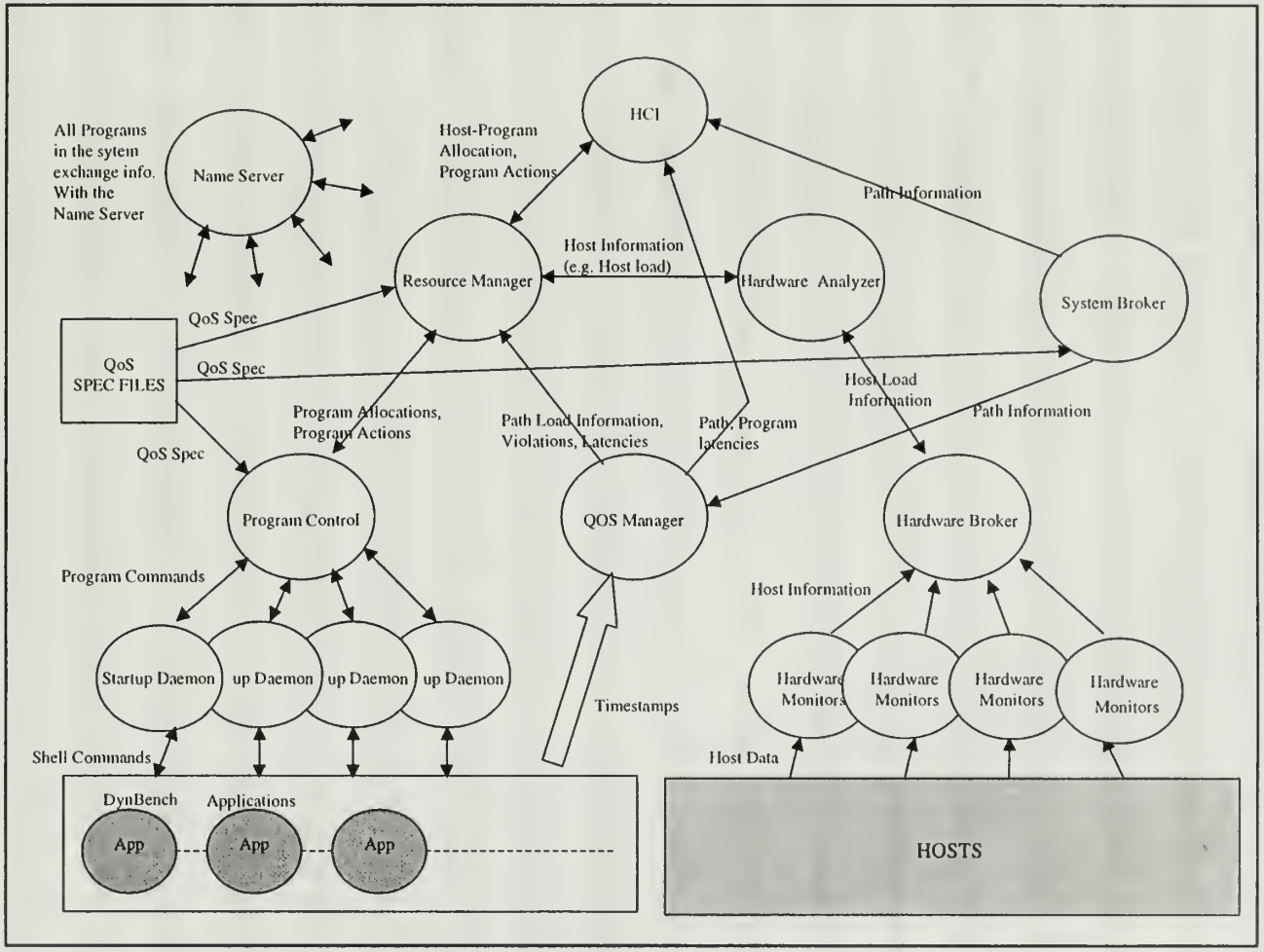

Figure 5. Desiderata software for resource and QoS management [DESI98]

The Resource Manager is activated when the dynamic paths are not meeting time constraints and when applications terminate abnormally. Resources are reallocated based upon these events to improve the QoS metrics of timeliness and fault tolerance. The Program Control component requests daemons on particular hosts to start and stop applications based upon mandates of the Resource Manager. Another function of the Program Control is to receive abnormal program termination information relayed to it by startup daemons and forward this information to the Resource Manager. The Startup Daemons reside only on one host and control applications' starts and stops. The Hardware Monitor is a daemon that resides on each host and is responsible for gathering load metrics for hosts and the LAN. It then passes this information to the Hardware Broker. The Hardware Broker collects all the monitors' metric information and develops an aggregate load index for each host, which is sent to the Hardware Analyzer - a component responsible for providing a sorted list of host load indexes. The QoS Manager(s) "detects if a real-time application path becomes unhealthy (misses its deadline), diagnoses the cause of the poor heath, and suggests corrective action (such as moving or replicating an application program) to the resource manager [DESI98]." The System Broker provides clients with information from the system specification and the Name Server maintains information on middleware application configuration. The final component of this system is the QoS Management Human Control Interface (HCI) that provides information to the user regarding configuration of the system, status and QoS of applications, and resources' reallocation operations. [DESI98]

C. QUALITY OF SERVICE REQUIREMENTS

1. Overview

To best maximize the use of existing resources, QoS metrics and monitoring are essential to resource management. An RMS must know the system's current status and the level of service it is responsible for delivering to the client. In a dynamic, combat environment, computing technology must be as adaptive as the soldier, and maintain situational awareness to execute mission-critical tasks. Though QoS specifications can be divided into policies and metrics, the performance metrics of timeliness, accuracy, and precision are best suited for the mission-critical and safety-critical systems used in military conflict [CHAT98].

2. Timeliness

Timeliness encompasses a class of metrics that measures time-related entities. It can be defined as a "representation of the timing requirements for performing a given piece of work [CHAT98]." Timeliness metrics are often quantified as latency, delay, or time to complete. They may also represent the earliest or latest start time for a task. For time-critical systems, timeliness is used synonymously with the term deadline. Failing to achieve a deadline in these types of applications may at a minimum, render collected data useless. At a maximum, it could result in the loss of life. In air defense systems utilizing the dynamic path paradigm, sensor data in the threat assessment path must be transferred to the engagement path. Also sensor data are given to the missile guidance path. The RMS must control the "timeliness" of these actions to ensure system synchronization, and threat detection and destruction.

3. Accuracy

The measure of data correctness and errors introduced into data by work and services performed define accuracy. [CHAT98] There must be a distinction made between data content and representation as the data flows through an application. Content refers to the accuracy of the data, whereas representation refers to the accuracy of the data's representation to the computer. If data generated at a certain level of accuracy cannot be depicted within the computer with that same level of accuracy, then applications lose the benefit of the data's initial correctness. Accuracy is especially important for an RMS in military applications because it is the responsibility of the RMS to "sell" information to its clients – decision-makers and operators. [CLAR97] In a manual system, inaccurate data can cause these clients to make uninformed and potentially disastrous decisions. In a system utilizing automated engagement, there is no "human factor" to aid in decision making, so inaccurate data will possibly cause the system to give an improper response.

4. Precision

Precision is also defined as it relates to content and representation. Precision of representation refers to the amount of data or work (volume)[CHAT98]. This volume is the physical amount of bits and bytes. The more bits used to represent a number, the greater number of decimal places are available for more precise calculations. To clarify the difference between accuracy and precision refer to Figure 4.

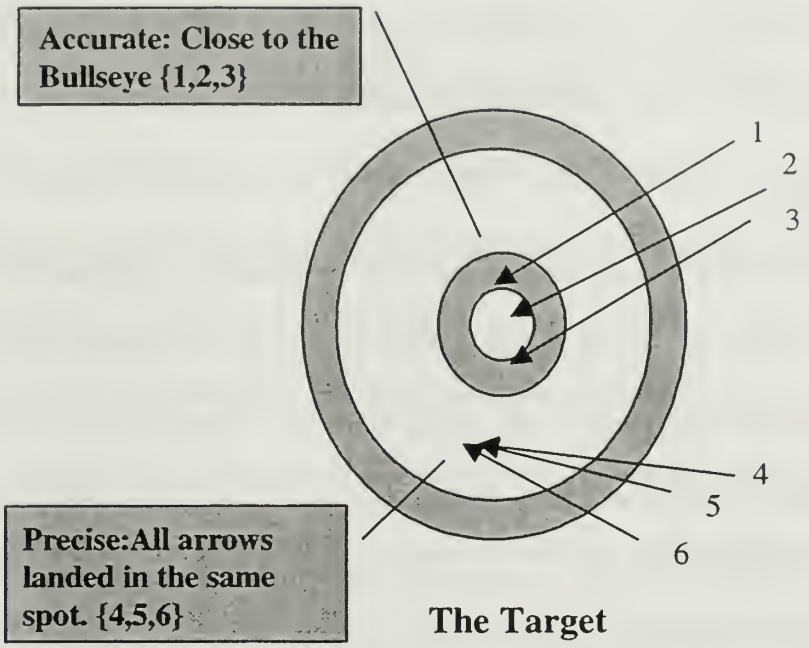

Figure 6. Precision vs. Accuracy

Using the ISO/OSI model, the precision of content in one layer translates into a precision of representation in the next lower layer. For example, a data packet in the network layer is composed of the payload and the header (content); the content of this data structure is understood by the network system. But the lower datalink layer recognizes that data only as a collection of data bytes and represents that as a series of bits. [CHAT98] The precision of representation affects how well the information can be read in the network layer of the destination host.

Air defense monitoring systems refer to precision as Track Quality (TQ). [CLAR97] TQ is calculated from sensor input and incorporated in the current track record. The TQ is then updated based on these calculations until the TQ (or precision)

drops to unacceptable levels and then the track is dropped. This is an example of how a RMS uses the precision metric in threat monitoring.

D. SUMMARY

Chapter II gave an overview of real-time, distributed applications and their QoS requirements. Section A focused on the AEGIS system, its components and functions. It analyzed the system's progression and applicability to command and control situations. Section B focused on the Desiderata project and how it applies to real-time distributed systems. It discussed the use of dynamic paths for QoS assessment and resource allocation, and how Desiderata could be used to further automate these actions. Section C presented and analyzed the QoS requirements of real-time distributed applications and how they can be measured. The QoS requirements were Timeliness, Accuracy, and Precision. The next chapter will discuss current RMS projects with a focus on MSHN and its relation to Desiderata.

III. RESOURCE MANAGEMENT TOOLS

Resource Management Systems (RMS) are commonly referred to as meta-computing systems though they actually build on top of the metacomputing framework. The RMS goal is to provide a prescribed quality of service (QoS) to processes and applications that are competing for distributed, heterogeneous resources [HENS99]. A RMS parallels the execution of a distributed operating system in that it views the group of computers it manages as a virtual machine [VAN 85] and attempts to provide the user with a "location-transparent" view of the resources. Through the use of a RMS, users should gain a higher level of resource availability and fault tolerance than would be possible on their local system alone [HENS99]. A RMS does not manage the resources of each computer, which is why it differs from a distributed operating system. The RMS is responsible for monitoring application, resource and QoS status across the virtual machine, and issuing commands to facilitate keeping those statuses at prescribed levels.

Chapter III discusses three on-going RMS research projects, their architectures and areas of focus. Section A details the GLOBUS [CZAJ97] project and its attempt to provide innovative resource management solutions. Section B provides an overview of the ERDOS [CHAT98] project and its resource management approach. Finally, Section C details the MSHN [HENS99] project as an adaptive QoS-driven RMS.

A. GLOBUS

Each attempt to implement resource management of geographically distributed resources begins with an identification of the problems to be solved. The GLOBUS project has identified five key elements of resource management that are addressed in its architecture.

- First, since resources are owned and operated by different organizations, in different domains, it is unlikely that there will be commonality in policies such as acceptable use, security, and scheduling. This is known as the *site autonomy* problem. [CZAJ97]
- Because each site is autonomous, it can be expected that each will use different local RMS or if the same RMS is used, it will be in a different and possibly incompatible configuration. This is called the *heterogeneous substrate* problem.[CZAJ97]
- The third problem is *policy extensibility*, which arises because heterogeneous distributed computing applications come from a wide range of domains with different resource requirements. Thus a RMS "must support the frequent development of new domain-specific management structures, without requiring changes to code installed at participating sites[CZAJ97]."
- Because some applications have requirements that can only be met though the use of several resources simultaneously, the fourth problem is *co-allocation.*

- The final problem of *online control* is very closely tied to QoS. In a RMS, negotiation is necessary to ensure that applications can adapt to resource availability, especially when requirements and resource status are dynamic.

GLOBUS' architecture was designed to address the five problems mentioned above. Entities known as *resource managers* confront the problems of heterogeneous substrate and site autonomy. The managers provide a well-defined interface to different local resource management tools, policies, and security implementations. The project has a *resource specification language* (RSL) that, in conjunction with *resource brokers,* is able to support negotiation between different components of a RMS, and handle application requests. These components address the problems of online control and policy extensibility. *Resource co-allocators* define various co-allocation strategies to defeat the co-allocation problem. Figure 5 presents the GLOBUS architecture. In this diagram, LSF, EASY-LL and NQE represent local resource schedulers and GRAM is the local resource manager. Key to this architecture is the *information service.* It is responsible for providing information about the current availability and service capabilities of resources. This information is used to: locate resources, determine resource properties, and process high-level resource specifications into specific manager requests [CZJA97]. A testbed for this architecture was developed that is comprised of 15 sites, 330 computers and 3600 processors [CZJA97].

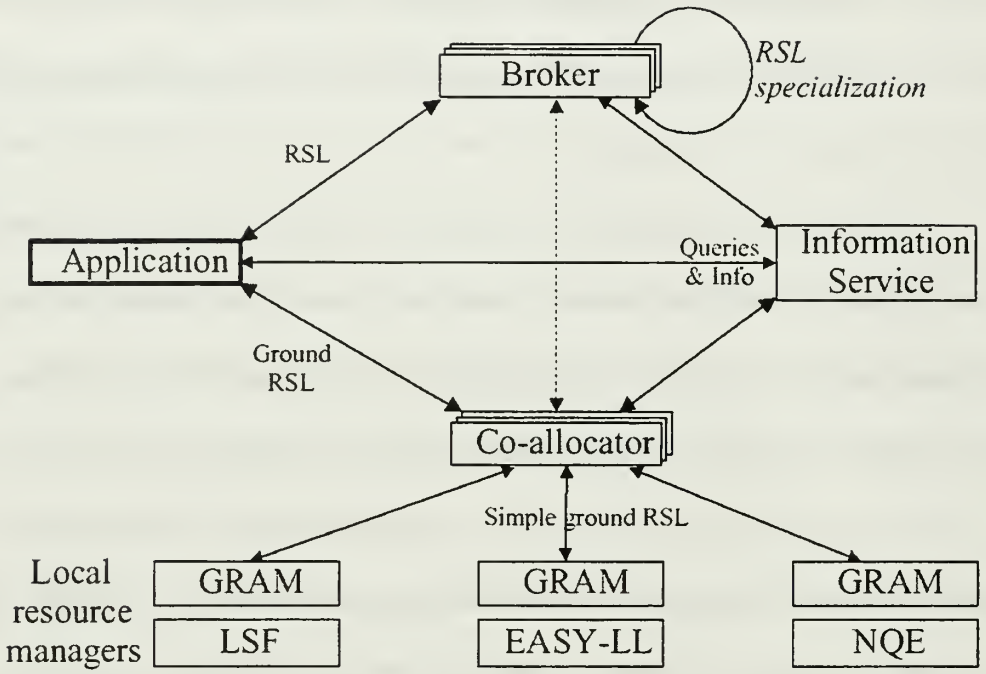

Figure 7. The Globus resource management architecture, showing how RSL specifications pass between application, resource brokers, resource co-allocators, and local managers. [CZJA97]

B. ERDOS

The End-to-End Resource Management of Distributed Systems (ERDOS) project is an architecture developed to provide adaptive, end-to-end, and scalable distributed resource management to applications [CHAT98]. This is a difficult goal to achieve especially in a distributed environment where real-time resource requirements dictate coordinated resource management. The problem is intensified by today's heterogeneous computing environment, high degree of resource sharing (GLOBUS' *co-allocation*

problem [CZJA97]) with varying QoS requirements, and a computing domain where resource availability and requirements are dynamically changing. [CHAT98]

The goal of ERDOS is to "identify the functions that must be preformed by each of the system layers in order to achieve end-to-end application-level QoS guarantees by performing QoS-driven resource management [CHAT98]." The following models capture information from the three perspectives (application, resource, and system) of resource management and enable the middleware to react to heterogeneous applications and resources:[CHAT98]

- Logical Application Stream Model (LASM): Application perspective – determines applications' structure in a system-independent manner.
- Benefit Function (BF): - An abstraction that models an application's QoS stipulations and preferences.
- Resource Model: Resource perspective – captures the resource information necessary for resource management algorithms.
- System Model (SM): System perspective – describes the layout and management structure of system resources.

ERDOS identifies the system layers as application, middleware, and resource. The architecture is not limited to proprietary applications or specific algorithmic policies. To facilitate communication between these layers, *application programmer interfaces* (APIs) are used. See Figure 8.

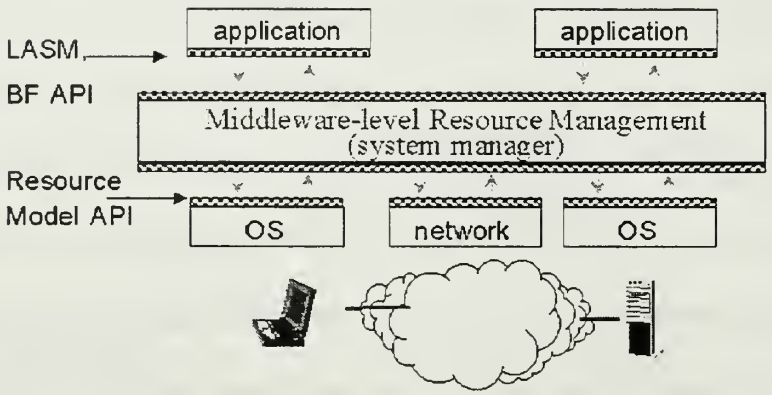

Figure 8. ERDOS System Architecture [CHAT98]

There are two primary API's in this architecture. The first API, the LASM-BF API, is between the middleware and the applications. The API facilitates the resource manager controlling the components of applications by determining on which resources the applications should run and at what level of QoS. This interface allows applications to run on different QoS-driven systems, independent of the RMS middleware implementations.[CHAT98]

The second API, the Resource Model API between the middleware and the resources, divides the middleware into generic and resource-specific parts. This division allows for masking of implementation details of resources from the middleware, and dealing with all resources in a uniform manner. The second point is especially important because it simplifies integration of new resource types.[CHAT98]

- The ERDOS middleware QoS architecture is shown in Figure 9 [CHAT98].

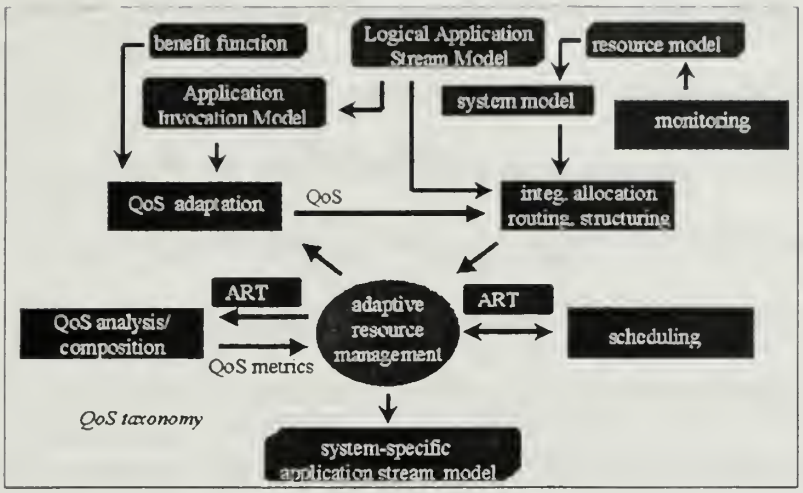

Figure 9. ERDOS Middleware QoS Architecture [CHAT98]

The middleware of this system encapsulates the crucial adaptive RMS algorithms that assign resources to applications, schedule applications on the shared resources, and adapt an application's QoS when system requirements exceed resources. The use of APIs provides portability to the system, which allows for the use of COTS software at the application, and resource layers. The adaptive nature of the RMS makes this system useful in a mission-critical or time-critical computing environment when QoS requirements are in conflict with system resources available.

C. MSHN

Another RMS envisioned for a heterogeneous computing environment is the *Management System for Heterogeneous Networks (MSHN).* The goal of MSHN, as with any real-time RMS, is to identify current resources available in a computing domain and allocate those resources to applications. Resource allocation mechanisms attempt to

provide a predetermined QoS to these applications based upon factors such as security, user preferences, and timeliness requirements. [HENS99]

MSHN is intended to be a research system to develop adaptive scheduling of process execution and provide support for *adaptive-aware* applications [HENS99]. Adaptive–aware refers to the ability of an application to input, process or output data in different versions based upon QoS metrics such as precision, accuracy and timeliness. In addition to resource allocation, MSHN's model describes adaptive mechanisms to allow for application migration from one system to another, if QoS violations are above an acceptable threshold.

"MSHN seeks to determine how to meet multiple different QoS requirements to multiple different applications simultaneously [HENS99]." There are two key issues that must be addressed in order to achieve this objective. First, a method must be developed to dynamically determine a value for a combination of QoS requirements. Second, resource allocation algorithms must match applications to resources that best achieve this value.

When a user requests service, a RMS should transparently locate resources in its computing domain to provide the service. The user should no be required to explicitly request the RMS to perform its task. MSHN's architecture (see Figure 10) provides for both while also addressing its objective's key issues.

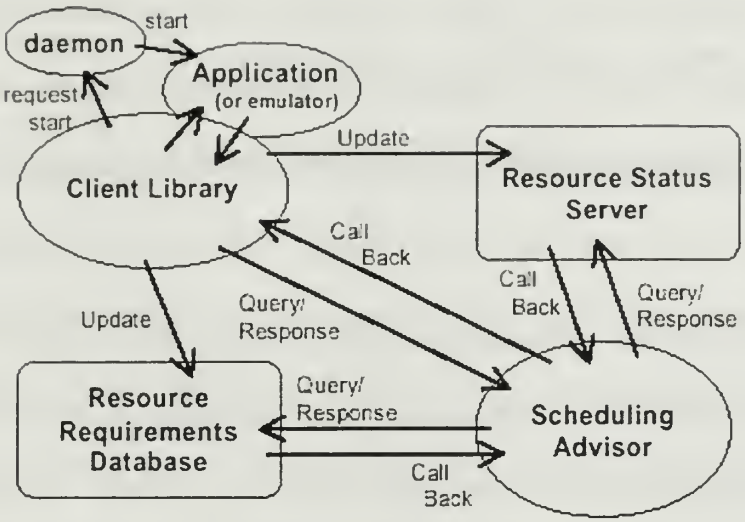

Figure 10. MSHN Conceptual Architecture [HENS99]

Central to MSHN's execution is the Client Library. The Client Library provides a mechanism for executing remote processes and for transparently determining the computing domain's resource status. The Client Library is "wrapped" around an application and intercepts that application's system calls [SCHN98]. Researchers at the University of Wisconsin refer to this action as *Process Hijacking* [ZAND97]. This technique is used for the RMS to gather application/process information on COTS software not designed to report to that RMS. A MSHN Daemon runs on each system to start processes on behalf of the Client Library.

The Scheduling Advisor's (SA) task is to make a best effort allocation of resources to applications. To achieve this, it receives information from the Resource Status Server (RRS) and the Resource Requirements Database (RRD). The RSS maintains a database of static, moderately dynamic, and highly dynamic resources. The RRD is a database that provides to MSHN a list of the resources necessary to execute

applications at their desired QoS. These components communicate with each other and are updated constantly to ensure proper operation of this RMS. Specifics of component execution and communication will be discussed in the following chapter.

D. SUMMARY

Chapter III provided an overview of three on-going RMS research projects, their architecture and areas of focus. Section A provided information on the GLOBUS project and its attempt provide innovative resource management solutions. Section B gave an overview of the ERDOS project and its current application towards resource management. Finally, Section C reviewed the MSHN project as an adaptive QoS-driven RMS. Chapter IV will analyze QoS status monitoring and QoS violation detection. The Desiderata and MSHN RMS projects will be used as case studies for this analysis.

IV. QOS VIOLATION MONITORING

Desiderata and MSHN are both QoS-driven RMSs that require QoS monitoring to effectively allocate resources. "Effectively", in this case, means *best effort* execution of an application to a user's requested and required QoS. Best effort execution gives the same priority to all applications, therefore, when the load is low, the RMS can deliver high-quality service easily. With an increased workload comes a uniform decrease in the service-quality levels the RMS is able to provide [HUST00]. For example, a user may indicate preferences for particular formats for output. A requested level of QoS may be streaming video, but if the resources required for video display are not available, then a text based representation may meet the QoS requirement. Although the user might not receive the most preferred format, the RMS working with the adaptable application is able to adjust and provide some level of service as opposed to none. In contrast, if the RMS is working with an application that is not adaptable, then QoS depends upon successful completion of the application. In a mission-critical system, where the key QoS metric is timeliness, the requested and required levels of QoS are synonymous. It is imperative that the RMS identify QoS violations, and react appropriately – for example, by migrating or by distributing processes to other hosts, or decreasing precision – so that requested and required levels can be met.

In this chapter, QoS violation monitoring techniques of RMSs will be reviewed. Section A presents the Desiderata process for monitoring QoS and detecting QoS violations. Section B presents our proposed MSHN QoS violation detection method.

A. DESIDERATA

QoS measurement has a major effect on Desiderata's operation. At the center of this measurement is the timestamp. The SendTimeStamp function's structure is as follows:

```
int SendTimeStamp(int   pathserv_sd,  //path manager's comm. socket
        char  Event[1],               //event parameter
        int   Sequence,               //current sequence number
        int   TacticalLoad,           //number of tracks-datastream
        char  Host[32],               //name of host of application
        char  ID[8],                  //process identification
        int   buffsize)               //size of  buffer or message size
```

This function is used to send a message to the Path Manager that conveys the current mode of an application. Pathserv_sd identifies the socket that applications should use to communicate with the Path Manager. The Event parameter takes the values of "R", "S", and "D", which signal to the Path Manger that an application is registering, starting or done respectively. Other parameters in the structure are the cycle count, host and process identification number of the sending program, and the buffer or message size. This function's return value indicates to the calling path application whether the Path Manager is Dead or Alive. The R, S, and D timestamp data are used by the QoS Manager in conjunction with the QoS specifications to determine whether a path is meeting its defined level of QoS. [SONA00] Load and violation information is then passed to the Resource Manager, which is responsible for program allocations and actions. These actions include spawning multiple versions of the same program or migrating a program

to a host with more resources available. Figure 11 displays the information and control flow of the Desiderata system. Note the timestamp information flow from applications to the QoS Manager.

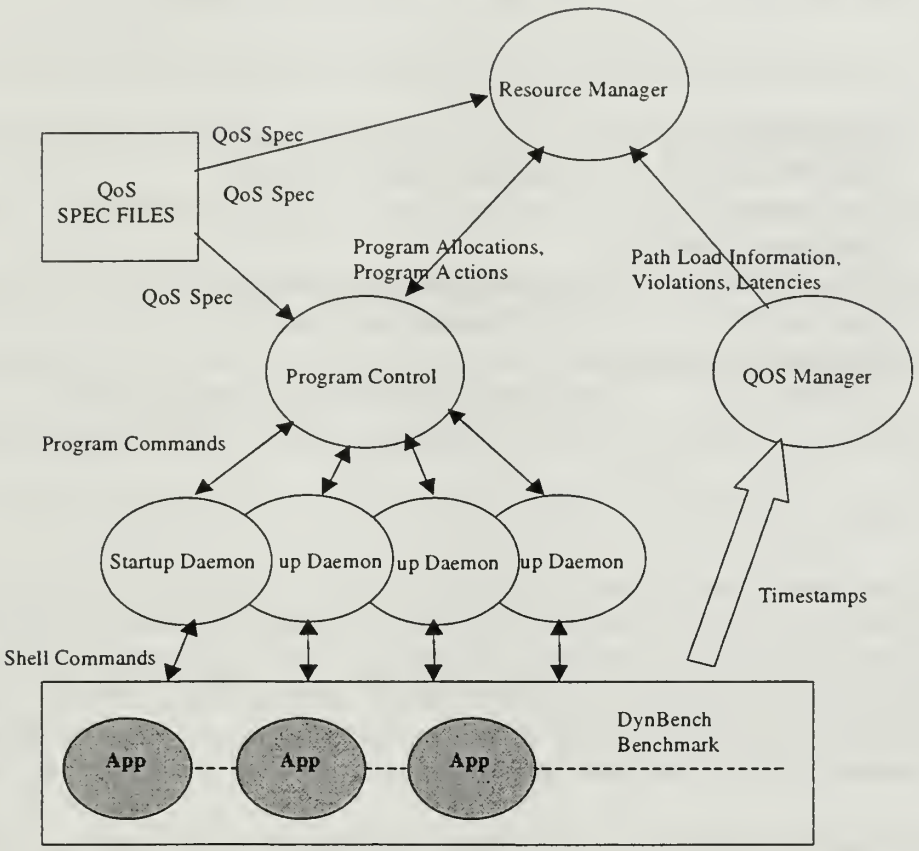

Figure 11. Timestamp Sequencing [DESI98]

Not pictured are the hosts where these programs are executed. To see how they are integrated review Figure 5. Application timeliness information is developed dynamically to assist in violation detection. It is compared to static specifications with the following Specfile syntax:

```
PATH Guidance{
 Connectivity{
  (D:B:MGM, D:B:MG); //This defines the path's flow – Missile Guidance
                        Manager to Missile Guidance System
 Type Continuous;        //This path is always executing.


 RealTimeQos{
  SimpleDeadline 1.0;  // Deadline for path execution
  BatchLatency 5.0;
  BatchInterArrival 5.0;
  Maxslack 80;          //Max deviation from deadline
  MinSlack 20;          //Min deviation from deadline
  SlidingWindowSize  20;
  Violations 15;        //Number of violations before flag is raised
        .
        .
        .
       etc
```

Though this is only a portion of one dynamic path specification in Desiderata, the other paths are very similar; the majority of a path's specification is devoted to stipulating time parameters such as the deadline and the minimum and maximum *slack* (deviation from the deadline) allowed during a path's execution. Though timeliness is addressed extensively, currently there are no aspects of this system that address areas of precision or accuracy.[SONA00]

B. MSHN

The vision of the MSHN project is to develop a system that can monitor QoS and dynamically manipulate processes and resources within its scope to provide the user with a requested level of QoS or an acceptable substitute. MSHN's Client Library (CL)

monitors application completion times by trapping the **exit** system call. The current implementation of the MSHN project comprehensively gathers data on an application's resource usage by "wrapping" that application, intercepting its system calls, and storing that data. With the monitoring data from these *discrete* applications, scheduling decisions can be made regarding the application termination actions necessary to achieve the requested QoS. The intent of this research is to adapt the CL to be able to monitor progress of *continuous* applications (those that do not include the **exit** system call), while maintaining the ability to evaluate those that are discrete. By adding a component to the MSHN system that is able to transparently detect continuous application QoS timeliness violations, MSHN will be able to perform its duties as a RMS with a wider range of legacy and COTS software. This is in contrast to Desiderata, which uses the SendTimeStamp, invoked via explicit function calls within the source code of the test applications (i.e., not transparent monitoring) to provide input to the QoS manager. The proposed MSHN component is the Transparent Path Monitoring System (TPMS).

1. Transparent Path Monitoring System(TPMS)

The TPMS is comprised of the following five modules:

- SpecDB Module -Specification Database
- PathDB Module -Path Database
- Path Timer Module

- Evaluate and Alert Module
- Control Module

The relationship of TPMS modules to MSHN modules is shown in Figure 12.

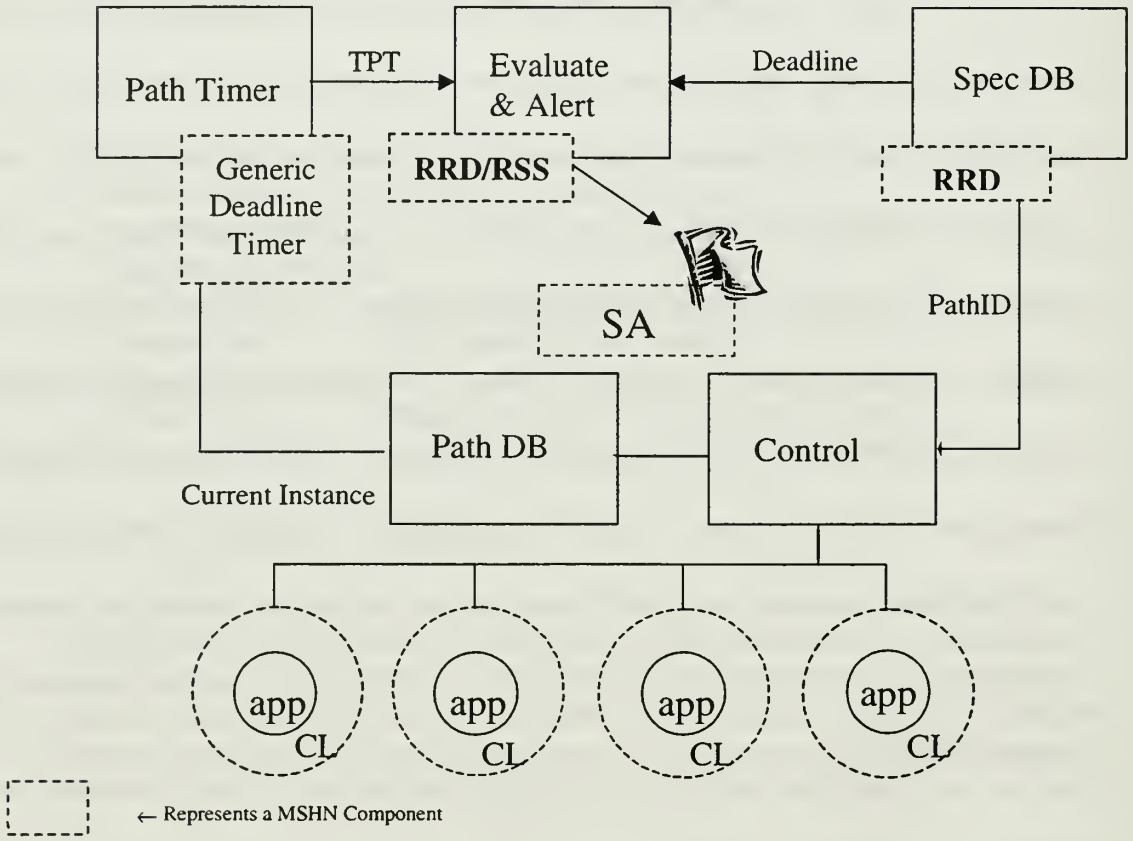

Figure 12: TPMS Representation in Regards to MSHN

The SpecDB Module is responsible for establishing in memory a database that contains path timeliness QoS specifications, which are similar to those used in Desiderata. This module parses a user-defined data file, and uses it to populate the database in memory; as a memory-resident database, there is fast access to data by the system. The specifications

detailed here include path identification number, length and deadline; the application identification numbers; and the order in which the application executes within a path. Once this module is initialized and populated, the TPMS initiates the PathDB module.

The PathDB module creates a PathDB database that contains the path identification number and path length for each path. It also contains pointers to the multiple instances of path status data that a single path may create. There will be a separate instance for each datastream. Each instance of that path contains an instance identification number, a counter to determine how many applications have reported, and an array that holds application initiation and completion times. These times are used to calculate path latency. The following depicts the results of PathDB module's initialization phase with one path, one instance and one application report time.

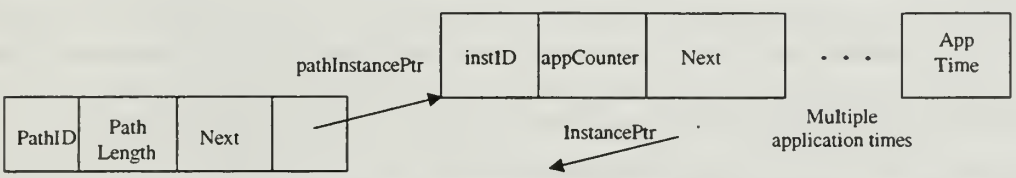

Figure 13: PathDB Module Initialization

In this case, pathLength and appCounter all have the value of "1". Instances of a path are created as necessary to accommodate application time data as they arrive. These times are placed in a dynamic array built at runtime. Its size is based upon the number of applications in a particular path. Because the applications used in this research are queue-based there is a possibility that a new timestamp for a single application may arrive at the

Control Module even though the prior path has not yet completed; therefore, multiple instances of the same path are supported. For example:

Sequence of Events	Action (Path A = Apps 1, 2, 3)	Result
1	Application 1 delivers timestamp	Stored in 1st instance of Path A
2	Application 1 delivers timestamp	Stored in 2nd instance of Path A
3	Application 2 delivers timestamp	Stored in 1st instance of Path A*

Table 1: Timestamp Sequencing

The assumption made in sequencing is that there is no possibility that a datastream passed to application 2 by the 1st instance of application 1 could be superseded by the datastream from the 2nd instance of application 1. Note that if you could show where the queue bottlenecks, then that particular element of the path would be a candidate for parallelization.

In the Evaluate and Alert Module, the Total Path Time (TPT) is compared to the deadline information stored in the Specification Database. If the TPT exceeds the specified deadline, then a flag is raised. It is then the responsibility of the MSHN's Scheduling Advisor to determine the best action to restore the system to the required level of QoS.

The Control Module is responsible for accepting applications' start and completion information from the CL. As an application executes, the CL sends the applications name and an event character- "S", starting and "D", done, periodically to the Control Module. With this information and a timestamp from the Control Module's host,

it populates the PathDB. Once all time positions are filled in a particular path instance, the TPMS conducts timeliness calculations and the system determines if a QoS violation has occurred. The specification for each module is provided in Appendix B.

2. MSHN Integration

For the TPMS to work during execution of path-based applications, it must receive an application ID and an event character indicating that an instance of a datastream has arrived at an application or that it has been passed to the next application. For this research, it was necessary to identify how communications are facilitated among the applications within the path. The testbed DynBench applications, which emulate an air defense system, establish sockets and send datastreams and messages using the `read()` and `write()` "C" programming functions. This is true whether applications are executed on a local or remote host. The `read()` and `write()` call interception of MSHN [SCHN98] were modified to send an event character and an applicationID to the TPMS Control Module. When an application receives a datastream though the `read()` function, the CL forwards to the TPMS the applicationID and the event character of "S", which indicates the application is starting to process data. The TPMS then logs the time it received the message. When calculations are complete on each distinct datastream, the application forwards the datastream to the next application in the path using the write() function. At this point the CL again sends the applicationID to the TPMS, this time with an event character of "D" to represent the completion of data processing. This completion time is now logged by the TPMS. Application latency is determined by calculating the difference between start and completion times of an application.

Similarly, calculating the difference between the start time of the first application and the completion time of the last application in a path provides the path latency or TPT.

The `read()` and `write()` functions are commonly used system calls; therefore, datastream receipt and forwarding must be distinguished from application-related usage. Applications that use end-to-end communication do so by establishing sockets and using the `read()` and `write()` functions. The client library has the ability to intercept system calls designed to create sockets and return their descriptors. The MSHN_sd_Class was developed to store and access these sockets' descriptors in a database. When a socket is created, its descriptor is inserted in the database. If a `read()` or `write()` call is intercepted, the file descriptor (fd) in the function call is compared against the MSHN_sd_Class database. If a match is found, the system call is executing a datastream receipt or forwarding event and the CL will send notification to the TPMS Control Module. Use of other file descriptors is ignored.

Since this implementation of the transparent QoS monitoring requires communications across multiple hosts, there must be some technique to access a common time reference. We have assumed that hosts within MSHN will not have perfectly synchronized clocks, so an implementation of the Network Time Protocol (NTP) was considered in past MSHN work. A remote process would query a timeserver and then use a formula to estimate clock offset and error [SCHN98]. In the current implementation, a User Datagram Protocol (UDP) server contained in the TPMS Control Module, accepts initiation and completion messages from the CL-wrapped applications communicating with UDP messages over the Internet Protocol(IP), and generates a

timestamp using the local clock time of the host. These messages along with the timestamps are recorded in the PathDB Module.

This method of determining application start and stop time of a unique datastream remedies the errors generated by host clocks that are not synchronized. We have assumed that path-based applications initiate and complete in the correct order, but because we have network communications, reporting of those actions may arrive in an incorrect order. Ongoing work may include CL-to-CL communications implemented to assign path-instance identifiers to the datastreams as they pass from application to application.

In future MSHN research, it is expected that the SpecDB Module will be incorporated into the RRD. Also the Evaluate and Alert Module's functionality will be incorporated into the RSS and RRD, with both updating the SA. Since a discrete application can be represented as a path of one, continuous as well as discrete applications will be supported by the enhanced MSHN system.

C. SUMMARY

Desiderata and MSHN are both QoS-driven RMSs that require QoS monitoring to allocate resources in a *best effort* execution of an application. In a mission-critical system, the requested and required levels of QoS are synonymous. It is imperative that the RMS identify QoS violations, and react appropriately – for example, by migrating or distributing processes to other hosts, or by decreasing precision. Section A presented the Desiderata process for monitoring QoS and detecting QoS violations. Section B presented a proposed MSHN QoS violation detection method described as the

Transparent Path Monitoring System (TPMS). Chapter V will detail experiments conducted and results obtained using MSHN's application wrapping technique with TPMS on Desiderata's DynBench air defense applications.

V. EXPERIMENTS AND RESULTS

A good monitoring tool's resource usage does not hamper the execution of the application or the system that it is monitoring. This chapter investigates the overhead of the TPMS implementation. TPMS receives information from path-based applications wrapped with the MSHN CL. In past MSHN work, it was determined that the CL adds an average of 3% overhead to each system call [SCHN98]. Since timeliness is the key QoS metric in mission-critical systems, path latency will be the focus of this investigation. This experiment will use the DynBench applications that emulate an air defense system with path-based programs. Section A details the experimental design necessary to capture overhead data. Section B gives the results of the experiments conducted.

A. EXPERIMENTAL DESIGN

Our experiments focused on path latency measurements of DynBench applications when run (1) unmonitored and unwrapped, (2) monitored by Desiderata RMS and (3) wrapped and integrated with TPMS. The hypothesis is that TPMS will increase a path's completion time, but the increase will be at an acceptable level and will not hamper DynBench's execution.

To better understand how DynBench applications work together see Figure 14.

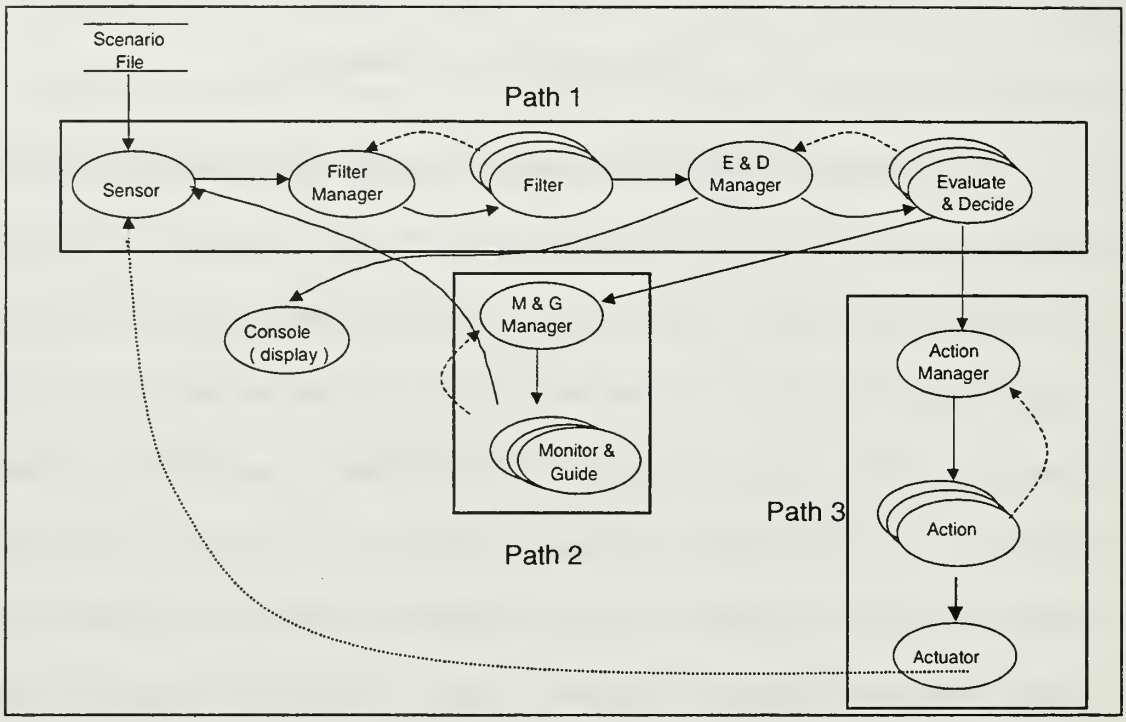

Figure 14. DynBench Application Communications [DESI98]

The scenario file gives initial input to the sensor simulating initial contact with aircraft and/or projectiles. The solid lines indicate transfer of a datastream or "track data" between applications. Applications shown as tiled in Figure 14 may be run as multiple instances to provide load sharing. Dashed lines indicate message traffic or a communication channel for simulated input. For example, the actuator program must send the positions of the missiles it has "fired" in response to threats to the sensor program. This updated track data is then passed throughout the system.

The hardware used in this experiment was an ULTRA 10 Unix workstation with 128MB RAM, and 300 MHz processor. The operating system is SUN® Solaris 5.5. Key to the path's completion time is the size of the data set used as input into the path. This is

significant because the presence of threat aircraft results in missiles being fired and a two-fold increase in the input data to the sensor program. To maintain a baseline, the same scenario file will be used for each run. This file contains the start position and movement vectors for each aircraft.

1. Violation Detection

The first experiment involves determining if the TPMS system can detect a QoS violation. Remembering that a QoS violation occurs when path latency exceeds the user defined deadline, it is necessary to determine what that deadline should be. The deadline for this experiment was determined statistically by calculating the standard deviation of a sample of Total Path Times (TPT) created with the TPMS. This resulted in the following data: (See Table 2.)

	AVG(MEAN) (secs)	STD DEV (secs)
TPT	0.121098	0.021053
Deadline	0.152678	

Table 2. Deadline Derivation

To calculate the deadline value, the standard deviation was multiplied by 1.5 and added to the arithmetic average (mean). With this calculation technique, the deadline provided enough violations to demonstrate the effectiveness of the system.

2. RMS Execution

The second experiment entails comparing path latency through the DynBench applications in an unmonitored mode, with Desiderata QoS management, and wrapped

and reporting to the TPMS. The unmonitored DynBench will be used as a baseline for comparison of the other RMS' implementations. To maintain consistency, all parameters such as track data, hardware, and OS will remain the same. This is an important test because it should demonstrate if there is a significant overhead increase between an RMS using transparent QoS detection (MSHN-TPMS) and an RMS detecting QoS using function calls embedded within monitored applications (Desiderata). Additionally, it will be important to note how much overhead TMPS adds to the basic application suite. Prior to testing, modifications to the DynBench applications were necessary to effect timeliness monitoring. A sample experiment was conducted to confirm that cycle time padding delays were restricted to the sensor application (first application in the path). The next phase was to determine the best place to insert probes to measure time and negate the effect of padding. These modifications had no effect on how the system ran, and did not skew latency data.

B. RESULTS OF EXPERIMENTS

1. Violation Detection

Execution of the first experiment was straightforward. The applications within the first DynBench path were wrapped and the path was given a deadline of 0.152678 seconds to complete. A few points must be noted. Track data are initialized in the sensor process using a data file called the scenario file. From there track data are generated by the sensor process; therefore, there is no direct input to the sensor program that could be caught, in a `read()` call, which would be used to alert the TPMS. Because

of this, the TPMS registers the start of the filter manager application (FM) as the start of the path.

By running 500 cycles of the path, it was found that 99.4% of the path completion times met this arbitrary deadline. This demonstrates that the TPMS supplemented with the CL can transparently detect QoS latency violations. But at what price do we achieve success? This forms the basis for the second experiment.

2. Effect of Monitoring on Path Latency

DynBench's unmonitored run (no Desiderata or MSHN management) at 1000, 2000 and 5000 cycles had total execution times of 20.426, 40.912 and 102.674 seconds. These times represented non-padded results and formed the baseline to which Desiderata and TPMS are compared.

The initial hypothesis was that TPMS monitoring would increase the execution time of the system due to its use of sockets to communicate to the CLs. This proved to be true. TPMS increased execution time in the 1000 and 2000 cycle runs by a factor of 0.007 seconds per cycle in comparison to the unmonitored DynBench run. It was 0.008 when the cycle total was 5000. Desiderata increased execution time by 0.002 seconds per cycle for each of the three cycle periods. See Table 3.

DynBench Variations	Unmonitored	w/Desiderata	Increase (secs/cycle)	w/TPMS and CL	Increase (secs/cycle)
1000 Cycles Completion Time (secs)	20.426	22.498	0.002	27.432	0.0070
2000 Cycles Completion Time	40.912	44.810	0.002	55.347	0.0072
5000 Cycles Completion Time (secs)	102.674	113.160	0.002	141.902	0.0078

Table 3. DynBench Execution Data

Table 3 demonstrates a uniform increase in execution time relative to cycle increase for all three versions of DynBench. Also, the execution time overages remained relatively consistent for each data set.

It is clear that Desiderata has less of an effect on path completion time than does TPMS. For TPMS, the CL interception of system calls and passing initiation and completion messages to the TPMS Control Module effects completion time of the applications within the path. The slight increase in seconds per cycle between unmonitored DynBench and TPMS upon cycle increase, indicates that there is some very small but cumulative delay introduced by TPMS as the test cycles increase. To further study this delay, we collected data on larger cycle runs of 10,000 for unmonitored DynBench and TPMS. Once again we had a slight increase in the seconds per cycle ratio. The most plausible reason for the increase is the multiple dynamic allocations and deallocations of memory for storing temporary data in the TPMS system. See chart in Figure 15 for comparison of overhead increases of Desiderata and TPMS.

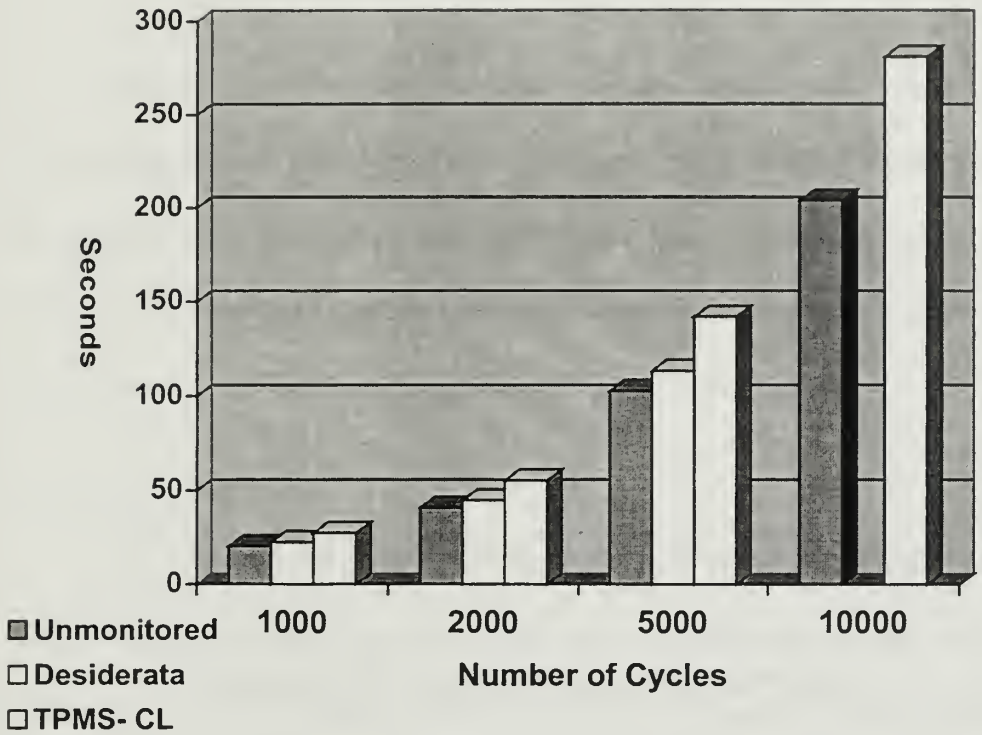

Figure 15. DynBench Overhead in Desiderata and TPMS

As TPMS is further integrated into the MSHN architecture, further study will be required to determine if the increased overhead is an acceptable tradeoff for using COTS applications and transparent QoS monitoring.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. CONCLUSIONS AND FUTURE WORK

This thesis accomplished the two primary objectives as outlined in Chapter I. First it describes, in detail, the QoS requirements for real-time, distributed applications based upon user criteria. Second, a technique was developed to transparently detect the QoS metric of timeliness within a path-based set of applications. This chapter discusses the contribution of this thesis and proposed follow-on work.

A. CONCLUSION

The Management System for Heterogeneous Networks (MSHN) requires usage information associated with applications that run within the MSHN system and status information for the resources within the MSHN scheduler's scope [SCHN98]. The scheduler makes decisions based upon this information. MSHN's Client Library (CL) is wrapped around an application and transparently gathers resource usage and application data. This thesis presented a technique to identify ongoing timeliness QoS violations through extensions to the CL.

The CL's task is to intercept system calls and send information to the MSHN system. Chapter IV presented a method, utilizing this interception capability, to monitor the timeliness QoS metric of path-based (i.e. continuous) applications. This method was the Transparent Path Monitoring System (TPMS). TPMS calculated the end-to-end path latency and compared it against a predetermined deadline provided by the end-user. If a

QoS violation occurred, a *flag* was raised within the system. Within the context of a complete RMS, a responsive action to a QoS violation could be initiated.

Though the TPMS could be added as an additional component to MSHN, it would be more appropriate to implement its functionality in MSHN's Scheduler, Resource Requirements Database (RRD) and Resource Status Server. Key to TPMS is its ability to monitor QoS without access to application source code. This lends itself to COTS or legacy software.

Testing of TPMS with the DynBench air defense simulation applications, proved the hypothesis that TPMS would increase path completion times of the Dynbench system. The CL's system call interception and reporting to the TPMS Control Module, increased the Total Path Time by an average of 0.007 seconds per path cycle. TPMS' integration into the MSHN architecture will require further study to determine if the increased overhead is an acceptable tradeoff for using COTS or legacy software whose QoS is transparently monitored.

B. FUTURE WORK

1. Expansion of Existing MSHN Wrapper Functionality

The development of the MSHN wrapper should proceed in two areas. In the current implementation, the MSHN CL is linked with the object code of the target application. The first area of research should be in development of a wrapping technique for executable code. The Executable Editing Library was developed at the University Wisconsin-Madison to accomplish similar goals [PORT99]. Also this topic is addressed

by Tim Fraser et. al. [FRAS99] in "Hardening COTS Software with Generic Software Wrappers." The second area of research should be focused on inter-wrapper communication. Currently each wrapped application communicates with the MSHN hierarchy but there may be some cases where applications need to "talk" to each other. Inter-CL communications provide the ability to tag data and trace its movement through MSHN's architecture.

2. Integration of Windows Application Monitoring

The DoD is migrating to a Windows based environment in most situations. In fact the Department of the Navy has decided that all future automation purchases will be based on the Windows NT operating system [CINC97]. Because of this, techniques for monitoring resources on Win32/x86 machines will need to be developed. Microsoft® is working on a research project called **Detours** which is a library for instrumenting arbitrary Win32 functions on x86 machines. This system intercepts Win32 functions by re-writing target function images [HUNT99]. This work closely corresponds to the work MSHN accomplishes in a UNIX environment. Additional thought should be applied to methods of applying MSHN to this alternate platform while continuing to adapt it to updated versions of the UNIX and LINUX operating systems.

3. Intergration of TPMS into MSHN

In future MSHN research, it is expected that the SpecDB Module will be incorporated into the Resource Requirements Database (RRD). Also the Evaluate and Alert Module's functionality will be incorporated into the Resource Status Server and RRD, with both updating and alerting the Scheduling Advisor. Since a discrete

application can be represented as a path of one, continuous as well as discrete applications will be supported by the enhanced MSHN system.

APPENDIX A. DATABASE DESIGN

This appendix provides a detailed description of the databases created in the Transparent Path Monitoring System's modules. The initialization data file exists in text format and can be modified with any text editor. The remaining databases are created dynamically at runtime and are stored in memory to facilitate fast access.

A. SPECIFICATION DATABASE

To initialize the system, the user must create a data file that contains application names and, path Ids, lengths, and deadlines. When the initialization function in the SpecDB module parses the user created data file, it then creates and populates the specification database dynamically. The fields are given in Table 4.

Field	Description
pathID	Uniquely identifies a path.
pathLength	Defines the number of applications in a path.
deadline	The time in which the path must complete.
next	A pointer that points to the next specification node.
appList	A linked list that contains the Application Identification in the order that they occur in the path.

Table 4. Specification Database

Figure 16 shows how the specification database logically exists in memory.

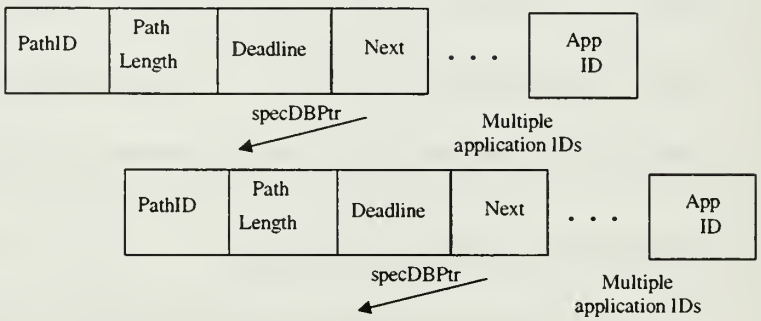

Figure 16. Specification Database in Memory

B. PATH DATABASE

Once populated, the specification database is static. In contrast the path database is constantly being updated with application timestamps, and the creation and deletion of path instances. Table 5 presents the fields of this database, and Table 6 details the fields of an instance node. An instance node maintains the application time information for a particular instance of a path. These instances are connected by means of a linked list and exist in memory only. The database is not stored between invocations of the TPMS. For graphical clarification, see Figure 13.

Field	Description
pathID	Uniquely identifies a path.
pathLength	Defines the number of applications in a path.
next	A pointer that points to the next path node.
instanceList	A linked list that contains multiple instances of the specified path.

Table 5. Path Database

Field	Description
instanceID	Uniquely identifies a path instance.
appCounter	Defines the number of applications that have reported to this instance of the path
next	A pointer that points to the next path instance.
timeArray	An array of length pathLength that is built dynamically and holds application timestamp information.

Table 6. Path Database Instance Definition

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B. TMPS MODULE SPECIFICATION

This appendix details the specifications of each of the TPMS modules. This specification includes information on included libraries, and structure definitions. Also public and private class functions are defined.

A. SPECDB MODULE

- LIBRARIES INCLUDED
  <assert.h> - for the assert function
  <iostream.h> - for file input
  <stddef.h> - for NULL
  <time.h> - for time operations

- TYPES DEFINED
  struct specDBrec - A cell/node in the list

  - DATA MEMBERS
    int pathID - path identification number
    int pathLength- the number of apps in the path
    float deadline - maximum path latency
    applicationID* applicationIDArray - An array that holds application names based on pathLength

- PRIVATE METHODS DEFINED
  PtrTo - Returns a pointer to the element in current position

- PUBLIC METHODS DEFINED
  specDBClass - Default constructor
  specDBClass - Copy constructor
  ~specDBClass - Destructor
  ListIsEmpty - Test to see if no elements exist in list
  ListLength - Number of items in list
  ListInsert - Put an item into the list at position
  ListDelete - Remove an item at position from list
  ListRetrieve - Get an item at a specific position
  = - Assignment one class object to another

DisplayList - Displays the list created
createSpecDB - Creates the specification database from an input file
appPositionInPath-Returns input application's position in its path
getDeadline -Gets deadline value for input path

B. PATHDB MODULE

1. pathDB.h

- LIBRARIES INCLUDED
  <iostream.h> - for I/O
  <assert.h> - for use with pointers
  "specDB.h" - to use specDBClass specDBrec
  "pathDBInstance.h" - to insert application times into path
  "evalalert" - to access goodInstance

- TYPES DEFINED
  struct pathDBrec - A cell/node in the list

  - DATA MEMBERS
    int pathID - path identification number
    int pathLength - the number of apps in the path
    instanceClass instanceList - list of instances of path

- PRIVATE METHODS DEFINED
  PtrTo - Returns a pointer to the element in the current position

- PUBLIC METHODS DEFINED
  pathDBClass - Default constructor
  pathDBClass - Copy constructor
  ~pathDBClass - Destructor
  ListIsEmpty - Test to see if no elements exist in list
  ListLength - Number of items in list
  ListInsert - Put an item into the list at position
  ListDelete - Remove an item at position from list
  ListRetrieve - Get an item at a specific position
  ListUpdate - Updates item at a specific position
  = - Assignment of one class object to another
  displayList - Displays the list created
  createPathDB - Creates the path database from SpecDB
  addAppTime - adds application time to path instance

2. pathDBInstance.h

- LIBRARIES INCLUDED
  <iostream.h> - for I/O
  <assert.h> - for use with pointers
  <time.h> - for time operations
  "specDB.h" - for access to specDBClass

- TYPES DEFINED
  struct instance - A cell/node in the list

  - DATA MEMBERS
    int instID - The identifier for instance
    int appCounter - Counts the number of reporting applications
    timeType timeArray[4] - An array that holds time information

- PRIVATE METHODS DEFINED
  PtrTo - Returns a pointer to the element in current position

- PUBLIC METHODS DEFINED
  instanceClass - Default constructor
  instanceClass - Copy constructor
  ~instanceClass - Destructor
  ListIsEmpty - Test to see if no elements exist in list
  ListLength - Number of items in list
  ListInsert - Put an item into the list at position
  ListDelete - Remove an item at position from list
  ListRetrieve - Get an item at a specific position
  ListUpdate - Updates item at a specific position
  = - Assignment of one class object to another
  displayInstanceList - Displays the list created
  instanceComplete - Determines if all time positions are filled in current instance

C. PATHTIMER MODULE

- LIBRARIES INCLUDED
  #include "pathDBInstance.h" - to access instanceClass list

- TYPES DEFINED
  None

- PRIVATE METHODS DEFINED
  None

- PUBLIC METHODS DEFINED
  calcTotalPathTime - Calculate total path time

D. EVALUATE AND ALERT MODULE

- LIBRARIES INCLUDED
- #include "specDB.h" - to access specDBClass list
- #include "pathDBInstance.h" - to access instanceClass list

- TYPES DEFINED
  None

- PRIVATE METHODS DEFINED
  None

- PUBLIC METHODS DEFINED
  goodInstance - Calculates to see if path made deadline
  evaluate - Checks to see if good instance
  raiseFlag - Raises a flag if path latency exceeds deadline

E. CONTROL MODULE

- LIBRARIES INCLUDED
- "specDB.h" - to access specDBClass list
- "pathDB.h" - to access pathDBClass list
- <stdio.h> - for I/O
- <stddef.h> - to define NULL
- <errno.h> - for errors
- <time.h> - to access time functions

- <sys/types.h> - to access socket connections
- <sys/socket.h> - same
- <netinet/in.h> - same
- <netdb.h> - same

- TYPES DEFINED
  None

- PRIVATE METHODS DEFINED
  None

- PUBLIC METHODS DEFINED
  None

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C. TPMS SOURCE CODE

A. SPECDB MODULE

```
/*****************************************************************
// FILE: specDB.cpp                                              *
//                                                               *
// AUTHOR: Kendal Polk as a modification of Template from Data   *
//         Abstraction and Problem solving with C++ by Frank     *
//         Carrano, CH4,pp165.                                   *
//                                                               *
// LAST MODIFIED: 6 June 2000                                    *
//                                                               *
// PURPOSE: This is the source file for operations necessary for the*
//          specDB Module                                        *
//                                                               *
//                                                               *
//****************************************************************/

#include "specDB.h"

#include <fstream.h>
#include <stddef.h>
#include <assert.h>
#include <strings.h>


struct specDBNode {
   specDBrec             item;
   specDBNodePtrType     next;
};

//****************************************************************
// SpecDBClass: Default constructor                              *
//****************************************************************

SpecDBClass::SpecDBClass(void): size(0), head(NULL)
{
}


//****************************************************************
// SpecDBClass: Copy constructor                                 *
//                                                               *
// INPUT: const   SpecDBClass& sourcelist - The source list to copy *
// LOCAL: specDBNodePtrType newPrev      - The new list head     *
//        specDBNodePtrType origCur      - local ptr             *
//                                                               *
//****************************************************************

SpecDBClass::SpecDBClass(const SpecDBClass& sourcelist):
size(sourcelist.size)
{
   specDBNodePtrType newPrev, origCur;
```

```
   if ( sourcelist.head == NULL )
      head = NULL;
   else {
      head = new specDBNode;
      assert(head != NULL);
      head->item = sourcelist.head->item;

      newPrev = head;

      for ( origCur =  sourcelist.head->next;
            origCur != NULL;
            origCur =  origCur->next ) {
         newPrev->next = new specDBNode;
         assert(newPrev->next != NULL);
         newPrev       = newPrev->next;
         newPrev->item = origCur->item;
      }

      newPrev->next = NULL;
   }
}// end



//*********************************************************************
// ~SpecDBClass: Destructor                                           *
//*********************************************************************

SpecDBClass::~SpecDBClass(void)
{
   int success;

   while ( !ListIsEmpty() )
      ListDelete(1, success);
}



//*********************************************************************
// ListIsEmpty: Checks if the list is empty.                          *
//                                                                    *
// RETURN: If list is empty the function returns TRUE, otherwise     *
//         it returns FALSE.                                          *
//                                                                    *
//*********************************************************************

int
SpecDBClass::ListIsEmpty(void)
{
   return(size == 0);
}
```

```
//*********************************************************************
// ListLength: Returns the number of items in the list                *
//                                                                    *
// RETURN: Returns the number of nodes in the list.                   *
//                                                                    *
//*********************************************************************

int
SpecDBClass::ListLength(void)
{
   return(size);
}



//*********************************************************************
// PtrTo: Find the requested node in the list.                        *
//                                                                    *
// INPUT:  int position - specifies the position of the requested     *
//                        node.                                       *
//                                                                    *
// RETURN: If successful, returns a pointer to the requested node,   *
//         otherwise it returns 0.                                    *
//                                                                    *
//*********************************************************************

specDBNodePtrType
SpecDBClass::PtrTo(int position)
{
   specDBNodePtrType trav;
   int               skip;

   if ( (position < 1) || (position > ListLength()) )
     return(NULL);
   else {
      trav = head;

      for ( skip = 1; skip < position; skip++ )
         trav = trav->next;

      return(trav);
   }
}
```

```
//*********************************************************************
// ListRetrieve: Gets data from specified node.                       *
//                                                                    *
// INPUT:  int   position - Position of node from which the data      *
//                         is to be retrieved.                        *
// LOCAL:  specDBNodePtrType cur      - pointer to the desired item   *
// OUTPUT: specDBrec&   dataItem      - The parameter into which the  *
//                                      desired item is retrieved     *
//         int&         success       - Returns TRUE if the retrieve  *
//                                      was successful, FALSE         *
//                                      otherwise                     *
//*********************************************************************
void
SpecDBClass::ListRetrieve(int position, specDBrec& dataItem,
                          int& success)
{
   specDBNodePtrType cur;

   success = ((position >= 1) &&
                            (position <= ListLength()));

   if ( success ) {
      cur      = PtrTo(position);
      dataItem = cur->item;
   }

   return;
}


//*********************************************************************
// ListInsert: Inserts new node into list.                            *
//                                                                    *
// INPUT:  int newPosition       - the position where to insert       *
//         specDBrec  newItem    - The item to insert                 *
// LOCAL:  int       newLength   - list length after insertion        *
//         specDBNocePtrType  newPtr   - node to hold new value       *
//         specDBNodePtrType    prev   - points to previous node for  *
//                                       reassigning pointers         *
// OUTPUT: int&            success     - TRUE if the insert was       *
//                                       successful, otherwise FALSE  *
//*********************************************************************

void
SpecDBClass::ListInsert(int newPosition, specDBrec newItem,
                        int& success)
{
   int     newLength;
   specDBNodePtrType newPtr, prev;

   newLength = ListLength() + 1;

   success =((newPosition >= 1) &&
                        (newPosition <= newLength));

   if ( success ) {
```

```
      size = newLength;

      newPtr  = new specDBNode;
      success =(newPtr != NULL);

      if ( success ) {
         newPtr->item = newItem;

         if ( newPosition == 1 ) {
            newPtr->next = head;
            head         = newPtr;
         } else {
            prev         = PtrTo(newPosition - 1);
            newPtr->next = prev->next;
                prev->next   = newPtr;
         }
      }
   }

   return;
}


//*********************************************************************
// ListDelete: Delete node from the list.                             *
// INPUT:  int   position - position of node to be deleted            *
// LOCAL:  specDBNodePtrType    cur    - saved pointer (to head or    *
//                                  next)                             *
//         specDBNodePtrType    prev   - pointer to the previous cell*
// OUTPUT: int&                 success  - TRUE if the delete was     *
//                                         successful FALSE otherwise*
//*********************************************************************

void
SpecDBClass::ListDelete(int position, int& success)
{
   specDBNodePtrType cur, prev;

   success = ((position >= 1) &&
                           (position <= ListLength()));

   if ( success ) {
      size--;

      if ( position == 1 ) {
         cur  = head;
         head = head->next;
      } else {
         prev       = PtrTo(position - 1);
         cur        = prev->next;
         prev->next = cur->next;
      }
      cur->next = NULL;
      delete cur;
      cur       = NULL;
   }
}
```

```
//*****************************************************************
// SpecDBClass: assignment =                                      *
//                                                                *
// INPUT: const   SpecDBClass& sourcelist - The source list to copy *
// LOCAL: specDBNodePtrType newPrev      - The new list head      *
//        specDBNodePtrType origCur      - local ptr              *
//                                                                *
//*****************************************************************

void
SpecDBClass::operator=(const SpecDBClass& sourcelist)
{

   specDBNodePtrType newPrev, origCur;
   size = sourcelist.size;
   if ( sourcelist.head == NULL )
      head = NULL;
   else {
      head = new specDBNode;
      assert(head != NULL);
      head->item = sourcelist.head->item;

      newPrev = head;

      for ( origCur =  sourcelist.head->next;
            origCur != NULL;
            origCur =  origCur->next ) {
         newPrev->next = new specDBNode;
         assert(newPrev->next != NULL);
         newPrev       = newPrev->next;
         newPrev->item = origCur->item;
      }

      newPrev->next = NULL;
   }
}

//*****************************************************************
// displayList: Displays the list created                         *
// LOCAL:  int    position - the position where to display node   *
//         int    success  - true when node retrieved             *
//         specDBrec DBrec       - the item to display            *
//*****************************************************************

void SpecDBClass::displayList()
{
      specDBrec         DBrec;
      int position = 1;
      int success;

      while(position <= ListLength())
      {
            ListRetrieve(position, DBrec, success);
```

```
            cout<<"\n Path "   <<    DBrec.pathID;

            cout<<"  Length " <<    DBrec.pathLength;

            cout<<" sec=> "<<DBrec.deadline.tv_sec<<", usec => ";
            cout<<DBrec.deadline.tv_usec<<endl;

            for (int appPos= 0;appPos < DBrec.pathLength; ++appPos)
            {
                  cout<<" App -  "<<DBrec.applicationIDArray[appPos];
            }
            position++;
      }//while
      cout<<endl;
}//end


//*************************************************************************
// createSpecDB: Builds the spec DB from input file                       *
// INPUT:  char*  inputfile   - the name of the file used to              *
//                              build database                            *
// LOCAL:  specDBrec DBrec    - Item to insert into the list              *
//         applicationID tempName[25]  - Used to hold name of app         *
//         int success     - true if node inserted in list                *
//                                                                        *
//*************************************************************************

void SpecDBClass::createSpecDB(char *inputfile)
{

      specDBrec DBrec;

      char tempName[25];

      int success;

      //open an input data file containing information on how the paths
      // are constructed
      // and deadline information

      ifstream specDBInputFile(inputfile,ios::in);

      //populate the Database
      while(specDBInputFile >> DBrec.pathID)
      {
            specDBInputFile >> DBrec.pathLength;

            specDBInputFile >> DBrec.deadline.tv_sec;

            specDBInputFile>> DBrec.deadline.tv_usec;

            DBrec.applicationIDArray = new
            applicationID[DBrec.pathLength];

            //populate the array with application IDs
            for (int count=0; count< DBrec.pathLength; count++)
            {
```

```
                specDBInputFile >>tempName;
                DBrec.applicationIDArray[count] = strdup(tempName);
            }


            ListInsert(ListLength()+1,DBrec,success);


      }//while

      specDBInputFile.close();

}//end


//*********************************************************************
// find PathID: Finds Path Identification number based on            *
//              application ID                                       *
// INPUT:  int application     - the number of the application       *
// LOCAL:  specDBrec  DBrec    - Item to search for a applicationID  *
//         int appNum          - Used to hold positions              *
//         int  specDBposition - Used to hold positions              *
//         int  success        - true if node inserted in list       *
//              found          - true if application found in path   *
// RETURN: the pathID of the input application                       *
//*********************************************************************

int SpecDBClass::findPathID(applicationID application)
{
      specDBrec   DBrec;

      int         appNum;

      int         specDBposition  = 1;

      int         success;

      int         found = 1;


      //Get a Path record and go the list of applications
      //to see if it is the correct one.

      while((specDBposition <= ListLength()) && (found !=0))
      {
            ListRetrieve(specDBposition, DBrec, success);

            int   appArrayPos = 0;

            while ((found!=0) && (appArrayPos < DBrec.pathLength) )
            {
            found =strcmp
            (DBrec.applicationIDArray[appArrayPos],application);
            appArrayPos++;
            }//while

      specDBposition++;//continue through all paths if necessary
```

```
      }//while

      return DBrec.pathID;

}//end

//*************************************************************************
// appPositionInPath: Finds application position in path                  *
// INPUT:  int  reportingApplication - the number of the application      *
//              specDBposition       - position in specDB                 *
// LOCAL:  specDBrec    DBrec        - Item to search for applicationID*
//         int  appArrayPos          - Used to hold positions             *
//         int  success              - true if node inserted in list      *
//              found                - true if reportingApplication is    *
//                                        found in a path                 *
// RETURN: the path location of the input application                     *
//*************************************************************************

int SpecDBClass::appPositionInPath(applicationID reportingApplication,
int specDBposition)

{
      specDBrec   DBrec;

      int         success;

      int         found = 1;

      int         appArrayPos = 0;


      //go to correct path based on specDBposition

      ListRetrieve(specDBposition, DBrec, success);


      //find application position in that path

      //strcmp function returns a zero if strings match

      while ((found != 0) && (appArrayPos < DBrec.pathLength) )
      {

            found = strcmp(DBrec.applicationIDArray[appArrayPos],
reportingApplication);


            //since Array enumeration is 1 less than logical position -
increase
            //appArrayPos even if found == true

            appArrayPos++;
      }


      // if entire array searched with no result then return a -1
```

```
      return(found==0 ? appArrayPos: -1);

}//end

//*********************************************************************
// getDeadline: Based on path ID returns deadline information          *
// INPUT:  int  path ID    - the number of the path                    *
// LOCAL:  specDBrec DBrec - Item to search for applicationID          *
//         int  success    - true if node inserted in list             *
//                                    found in a path                  *
// RETURN: the path deadline information                               *
//*********************************************************************

timeType SpecDBClass::getDeadline(int pathID)
{
      specDBrec   tempDBrec;
      int         success;

      ListRetrieve(pathID,tempDBrec,success);

      return(tempDBrec.deadline);

}//end
```

B. PATHDB MODULE

1. PathDB Source Code

```
/*********************************************************************
// FILE: pathDB.cpp                                                   *
//                                                                    *
// AUTHOR: Kendal Polk as a modification of Template from Data        *
//         Abstraction and Problem solving with C++ by Frank          *
//         Carrano, CH4,pp165.                                        *
//                                                                    *
// LAST MODIFIED: 6 June 2000                                         *
//                                                                    *
// PURPOSE: This is the source file for operations necessary for the  *
//          pathDB Module                                             *
//                                                                    *
//                                                                    *
//*********************************************************************/

#include <iostream.h>
#include <fstream.h>
#include <stddef.h>
#include <assert.h>
#include "pathDB.h"
#include "pathTimer.h"
#include "pathDBInstance.h"

struct pathDBNode {
   pathRec              item;
   pathDBNodePtrType    next;
```

```
};

//*********************************************************************
// pathDBClass: Default constructor                                   *
//*********************************************************************

pathDBClass::pathDBClass(void): size(0), head(NULL)
{
}


//*********************************************************************
// pathDBClass: Copy constructor                                      *
//                                                                    *
// INPUT: const   pathDBClass& sourcelist - The source list to copy  *
// LOCAL: ptrType newPrev      - The new list head                    *
//        ptrType origCur      - local ptr                            *
//                                                                    *
//*********************************************************************

pathDBClass::pathDBClass(const pathDBClass& sourcelist):
size(sourcelist.size)
{
   pathDBNodePtrType newPrev, origCur;

   if ( sourcelist.head == NULL )
      head = NULL;
   else {
      head = new pathDBNode;
      assert(head != NULL);
      head->item = sourcelist.head->item;

      newPrev = head;

      for ( origCur =  sourcelist.head->next;
            origCur != NULL;
            origCur =  origCur->next ) {
         newPrev->next = new pathDBNode;
         assert(newPrev->next != NULL);
         newPrev       = newPrev->next;
         newPrev->item = origCur->item;
      }

      newPrev->next = NULL;
   }
}// end



//*********************************************************************
// ~pathDBClass: Destructor                                           *
//*********************************************************************

pathDBClass::~pathDBClass(void)
{
   int success;
```

```
   while ( !ListIsEmpty() )
      ListDelete(1, success);
}



//*********************************************************************
// ListIsEmpty: Checks if the list is empty.                          *
//                                                                    *
// RETURN: If list is empty the function returns TRUE, otherwise     *
//         it returns FALSE.                                          *
//                                                                    *
//*********************************************************************

int
pathDBClass::ListIsEmpty(void)
{
   return(size == 0);
}



//*********************************************************************
// ListLength: Returns the number of items in the list                *
//                                                                    *
// RETURN: Returns the number of nodes in the list.                   *
//                                                                    *
//*********************************************************************

int
pathDBClass::ListLength(void)
{
   return(size);
}



//*********************************************************************
// PtrTo: Find the requested node in the list.                        *
//                                                                    *
// INPUT:  int position - pathifies the position of the requested     *
//                        node.                                       *
//                                                                    *
// RETURN: If successful, returns a pointer to the requested node,   *
//         otherwise it returns 0.                                    *
//                                                                    *
//*********************************************************************

pathDBNodePtrType
pathDBClass::PtrTo(int position)
{
   pathDBNodePtrType trav;
   int                        skip;
```

```
   if ( (position < 1) || (position > ListLength()) )
     return(NULL);
   else {
      trav = head;

      for ( skip = 1; skip < position; skip++ )
         trav = trav->next;

      return(trav);
   }
}


//*************************************************************************
// ListRetrieve: Gets data from specified node.                           *
//                                                                        *
// INPUT:  int    position - Position of node from which the              *
//                           data is to be retrieved.                     *
//                                                                        *
// LOCAL:  pathDBNodePtrType cur      - pointer to the desired item       *
// OUTPUT: pathDBrec& dataItem - The parameter into which the             *
//                               desired item is retrieved                *
//         int&   success  - Returns TRUE if the retrieve                 *
//                           was successful, FALSE                        *
//                           otherwise                                    *
//*************************************************************************

void
pathDBClass::ListRetrieve(int position, pathRec& dataItem,
                          int& success)
{
   pathDBNodePtrType cur;

   success = ((position >= 1) &&
                           (position <= ListLength()));

   if ( success ) {
      cur      = PtrTo(position);
      dataItem = cur->item;
   }

   return;
}
```

```
//*************************************************************************
// ListUpdate: Updates data from specified node.                          *
//                                                                        *
// INPUT:  int    position - Position of node from which the              *
//                           data is to be updated.                       *
//                           Reference to data-item to                    *
//                           which the data from the node is to be        *
//                           copied.                                      *
//         pathDBrec& dataItem - The parameter into which the             *
//                               desired item is retrieved/updated.       *
//                                                                        *
// LOCAL:  pathDBNodePtrType cur      - pointer to the desired item       *
//                                                                        *
// OUTPUT: int&   success  - Returns TRUE if the update was successful    *
//                           FALSE otherwise                              *
//*************************************************************************

void
pathDBClass::ListUpdate(int position, pathRec& dataItem,
                        int& success)
{
   pathDBNodePtrType cur;

   success = ((position >= 1) &&
                           (position <= ListLength()));

   if ( success ) {
      cur      = PtrTo(position);
      cur->item = dataItem;
   }

   return;
}

//*************************************************************************
// ListInsert: Inserts new node into list.                                *
//                                                                        *
// INPUT:  int newPosition    - the position where to insert              *
//                              the new node                              *
//         pathDBrec newItem  - The item to insert                        *
//                                                                        *
// LOCAL:  int newLength      - list length after insertion               *
//         pathDBNocePtrType  newPtr  - node to hold new value            *
//         pathDBNodePtrType  prev    - points to previous node for       *
//                                      reassigning pointers              *
//                                                                        *
// OUTPUT: int& success     - TRUE if the insert was successful,          *
//                            otherwise FALSE                             *
//                                                                        *
//*************************************************************************

void
pathDBClass::ListInsert(int newPosition, pathRec newItem,
                    int& success)
{
   int     newLength;
   pathDBNodePtrType newPtr, prev;
```

```
   newLength = ListLength() + 1;

   success =((newPosition >= 1) &&
                    (newPosition <= newLength));
   if ( success ) {
      size = newLength;

      newPtr  = new pathDBNode;
      success =(newPtr != NULL);

      if ( success ) {
         newPtr->item = newItem;

         if ( newPosition == 1 ) {
            newPtr->next = head;
            head         = newPtr;
         } else {
            prev         = PtrTo(newPosition - 1);
            newPtr->next = prev->next;
                prev->next   = newPtr;
         }
      }
   }

   return;
}

//*************************************************************************
// ListDelete: Delete node from the list.                                 *
//                                                                        *
// INPUT:  int position - position of node to be deleted.                 *
// LOCAL:  pathDBNodePtrType  cur - saved pointer (to head or next)       *
//         pathDBNodePtrType  prev- pointer to the previous cell          *
// OUTPUT: int&   success  - TRUE if the delete was successful,           *
//                           FALSE otherwise.                             *
//                                                                        *
//*************************************************************************

void
pathDBClass::ListDelete(int position, int& success)
{
   pathDBNodePtrType cur, prev;

   success = ((position >= 1) &&
                           (position <= ListLength()));

   if ( success ) {
      size--;

      if ( position == 1 ) {
         cur  = head;
         head = head->next;
      } else {
         prev       = PtrTo(position - 1);
         cur        = prev->next;
         prev->next = cur->next;
```

```
      }

      cur->next = NULL;
      delete cur;
      cur       = NULL;
   }
}

//*********************************************************************
// pathDBClass: assignment =                                          *
//                                                                    *
// INPUT: const   pathDBClass& sourcelist - The source list to copy *
// LOCAL: pathDBNodePtrType newPrev      - The new list head         *
//        pathDBNodePtrType origCur      - local ptr                 *
//                                                                    *
//*********************************************************************

void
pathDBClass::operator=(const pathDBClass& sourcelist)
{

   pathDBNodePtrType newPrev, origCur;
   size = sourcelist.size;
   if ( sourcelist.head == NULL )
      head = NULL;
   else {
      head = new pathDBNode;
      assert(head != NULL);
      head->item = sourcelist.head->item;

      newPrev = head;

      for ( origCur =  sourcelist.head->next;
            origCur != NULL;
            origCur =  origCur->next ) {
         newPrev->next = new pathDBNode;
         assert(newPrev->next != NULL);
         newPrev       = newPrev->next;
         newPrev->item = origCur->item;
      }

      newPrev->next = NULL;
   }
}


//*********************************************************************
// displayList: Displays the list created                             *
// LOCAL:  int          position - the position where to dispay node *
//                      success  - true when node retrieved form list*
//         pathDBrec    DBrec       - the item to display           *
//*********************************************************************

void pathDBClass::displayPathList()
{
      pathRec           pathDBrec;
```

```
      int position = 1;
      int success;

      while(position <= ListLength())
      {
            ListRetrieve(position, pathDBrec, success);

            cout<<"\n Path "  <<    pathDBrec.pathID;

            cout<<"  Length " <<    pathDBrec.pathLength;

            pathDBrec.instanceList.displayInstanceList();

            position++;
      }//while
}//end
```

```
//*********************************************************************
// createPathDB: Builds the spec DB from input file                   *
// INPUT:  char* inputfile      - the name of the file used to        *
//                                build the  database                 *
// LOCAL:  pathrec tempPath     - Item to insert into the list        *
//         specDBRec DBrec - Used to hold temporary information      *
//         int success      - true if node inserted in list          *
//             position     - retrieval point                         *
//*********************************************************************

void pathDBClass::createPathDB(SpecDBClass inputList)
{

      pathRec           tempPath;

      specDBrec   DBrec;

      int         success;

      int               position = 1;

      while(position <= inputList.ListLength())
      {
            inputList.ListRetrieve(position, DBrec, success);

            tempPath.pathID = DBrec.pathID;

            tempPath.pathLength = DBrec.pathLength;
```

```
            position++;

            ListInsert(ListLength()+1,tempPath,success);

      }//while

}//end


//*********************************************************************
// addAppTime: Inserts new time in instance list.                     *
// INPUT:  char  appStatus - signals start or stop of an application *
//         applicationID  reportingApplication - reporting app number*
//         timeval      currentTimePtr- pointer to current time value*
//         specDBList   inputList     - accessed to determine        *
//                                      deadline                     *
// LOCAL: int        PathID          - ID of path                     *
//                   instanceInsertionPoint- shows where to place in *
//                                           list                    *
//                   pathPosition    - identifies app position in path*
//                   currentPosition- counter for moving through list*
//                   timeArrayLength- length of array of times       *
//                   totalInstances - used to compare against counter*
//                   timeArrayInsert- position to insert in timearray*
//                   success         - used as a boolean             *
//                   instanceDone    - used as a boolean             *
//                   timeInserted    - used as a boolean             *
//                                                                   *
//         pathDBrec tempPathRec     - temporary path DB record      *
//                                                                   *
//         specDBrec tempDBrec       - temporary spec DB record      *
//                                                                   *
//         timeType  TPT             - total path time               *
//                   deadline        - deadline for the path         *
//                                                                   *
//         instance  firstInstance   - used to create an instance    *
//                   currentInstance- used to manipulate an instance *
//                                                                   *
//         pathDBNodePtrType         updatePathPtr- walks down path  *
//                                   list                            *
//*********************************************************************

void pathDBClass::addAppTime(char appStatus,
            applicationID reportingApplication,
              timeval* currentTimePtr, SpecDBClass inputList)

{
      int   PathID,     instanceInsertionPoint, pathPosition;

      int   currentPosition = 1;

      int   timeArrayLength, totalInstances,timeArrayInsert;

      int   success, instanceDone;

      int   timeInserted = 0;
```

```
pathRec     tempPathRec;

specDBrec   tempDBrec;

timeType    TPT,deadline;

instance    firstInstance,currentInstance;

pathDBNodePtrType updatePathPtr;


//find the Path the application belongs to

PathID = inputList.findPathID(reportingApplication);

//get correct path
updatePathPtr = PtrTo(PathID);

//determine how many applications in the Path
timeArrayLength= updatePathPtr->item.pathLength - 1;


//find the position of the application in the path

pathPosition = inputList.appPositionInPath
            (reportingApplication, PathID);

  #ifdef TPMS_DEBUG
   cout<<"pathPosition = "<<pathPosition<<"\n";
  #endif

//is this the first application of the path?

if((pathPosition == 1)&&(appStatus == 'D'))
{

      //if this is the first application of the path  AND it is
      // reporting that it is starting then
      //create an instance and put this time in it.

      // create an instance

      cout<<"creating new Instance\n";

      instanceInsertionPoint =  updatePathPtr->
         item.instanceList.ListLength()+1;


      success=(firstInstance.timeArray != NULL);

          #ifdef TPMS_DEBUG
             if (success) cout<<"success with time"<<endl;
          #endif

      //correct for a negative fraction of second
      if(currentTimePtr->tv_usec < 0){
```

```
                if(currentTimePtr->tv_sec >= 0){
                 currentTimePtr->tv_sec -= 1;
                 currentTimePtr->tv_usec += 1000000;
               }
               else{
                  currentTimePtr->tv_usec *= -1;
               }//end if-else
            }
            else if(currentTimePtr->tv_sec < 0){
                currentTimePtr->tv_sec  += 1;
                currentTimePtr->tv_usec = 1000000 -
                currentTimePtr->tv_usec;
            }//end if

      firstInstance.timeArray[0]= *currentTimePtr;

      firstInstance.appCounter = 1;


      updatePathPtr->item.instanceList.ListInsert
                  (instanceInsertionPoint,firstInstance,success);


}//end if

else if (pathPosition /= 1)
//this is not the first application and we must find the location
// for this application time IF it is a application complete time
{

   totalInstances = updatePathPtr->
   item.instanceList.ListLength();

        #ifdef TPMS_DEBUG
              cout<<totalInstances<<" = TotalInst "<<"\n";
        #endif

      while((currentPosition <= totalInstances) && !timeInserted)
      {
            //get the list of instances from the correct path

            updatePathPtr->item.instanceList.ListRetrieve
                        (currentPosition,currentInstance,success);


            //determine if the app time goes to this instance

            if ((pathPosition - currentInstance.appCounter)==1)
            {

                  //place this application time in the correct
                        //time position


                  timeArrayInsert  = currentInstance.appCounter;
```

```
                        currentInstance.timeArray[ timeArrayInsert ]=
                          *currentTimePtr;

                        currentInstance.appCounter++;

                        timeInserted = 1;

                        //Update new list information
                        updatePathPtr->item.instanceList.ListUpdate
                              (currentPosition,currentInstance,success);

                        instanceDone = updatePathPtr->
                              item.instanceList.instanceComplete
                    (updatePathPtr->item.pathLength,currentInstance);


                        if (instanceDone)
                        {
                              TPT = calcTotalPathTime(currentInstance);

                        #ifdef TPMS_DEBUG
                              cout<<" I am here in instanceDone\n";
                        #endif

                              deadline = inputList.getDeadline(PathID);

                              evaluate(deadline,TPT);

                              updatePathPtr->item.instanceList.
                                    ListDelete(1,success);
                        }//if
                  }//end if

                  if (!timeInserted)
                      currentPosition++;

                      //continue to next position in list
                      //if location for time not found here


            }//while

      }//if-else


}//end
```

2. PathDBInstance Source Code

```
/*****************************************************************
// FILE: pathDBInstance.cpp                                       *
//                                                                *
// AUTHOR: Kendal Polk as a modification of Template from Data    *
//         Abstraction and Problem solving with C++ by Frank      *
//         Carrano, CH4,pp165.                                    *
//                                                                *
// LAST MODIFIED: 6 June 2000                                     *
//                                                                *
// PURPOSE: This is the source file for operations necessary for the*
//          pathDBInstance Module                                 *
//                                                                *
//                                                                *
//*****************************************************************/

#include "specDB.h"
#include <stddef.h>
#include <assert.h>
#include "pathDBInstance.h"


struct instanceNode {
   instance                  item;
   instanceNodePtrType  next;
};




//*****************************************************************
// pathDBClass: Default constructor                               *
//*****************************************************************

instanceClass::instanceClass(void): size(0), head(NULL)
{
}




//*****************************************************************
// pathDBClass: Copy constructor                                  *
//                                                                *
// INPUT: const   pathDBClass& sourcelist - The source list to copy *
// LOCAL: ptrType newPrev      - The new list head                *
//        ptrType origCur      - local ptr                        *
//                                                                *
//*****************************************************************

instanceClass::instanceClass(const instanceClass& sourcelist):
size(sourcelist.size)
{
```

```
   instanceNodePtrType newPrev, origCur;

   if ( sourcelist.head == NULL )
      head = NULL;
   else {
      head = new instanceNode;
      assert(head != NULL);
      head->item = sourcelist.head->item;

      newPrev = head;

      for ( origCur =  sourcelist.head->next;
            origCur != NULL;
            origCur =  origCur->next ) {
         newPrev->next = new instanceNode;
         assert(newPrev->next != NULL);
         newPrev       = newPrev->next;
         newPrev->item = origCur->item;
      }

      newPrev->next = NULL;
   }
}// end



//*********************************************************************
// ~pathDBClass: Destructor                                           *
//*********************************************************************



instanceClass::~instanceClass(void)
{
   int success;

   while ( !ListIsEmpty() )
      ListDelete(1, success);
}


//*********************************************************************
// ListIsEmpty: Checks if the list is empty.                          *
//                                                                    *
// RETURN: If list is empty the function returns TRUE, otherwise     *
//         it returns FALSE.                                          *
//                                                                    *
//*********************************************************************


int
instanceClass::ListIsEmpty(void)
{
   return(size == 0);
}
```

```
//*********************************************************************
// ListLength: Returns the number of items in the list                *
//                                                                    *
// RETURN: Returns the number of nodes in the list.                   *
//                                                                    *
//*********************************************************************

int
instanceClass::ListLength(void)
{
   return(size);
}

//*********************************************************************
// PtrTo: Find the requested node in the list.                        *
//                                                                    *
// INPUT:  int position - identifies the position of the requested    *
//                        node.                                       *
//                                                                    *
// RETURN: If successful, returns a pointer to the requested node,   *
//         otherwise it returns 0.                                    *
//                                                                    *
//*********************************************************************

instanceNodePtrType
instanceClass::PtrTo(int position)
{
   instanceNodePtrType trav;
   int                         skip;

   if ( (position < 1) || (position > ListLength()) )
     return(NULL);
   else {
      trav = head;
      for ( skip = 1; skip < position; skip++ )
         trav = trav->next;
      return(trav);
   }
}



//*********************************************************************
// ListRetrieve: Gets data from specified node.                       *
//                                                                    *
// INPUT:  int           position - Position of node from which the   *
//                                  data is to be retrieved.          *
//                                                                    *
// LOCAL:  ptrType       cur      - pointer to the desired item      *
// OUTPUT: instance& dataItem - The parameter into which the          *
//                                  desired item is retrieved         *
//         int   &       success  - Returns TRUE if the retrieve     *
//                                  was successful, FALSE             *
//                                  otherwise                         *
//*********************************************************************
```

```
void
instanceClass::ListRetrieve(int position, instance& dataItem,
                        int& success)
{
   instanceNodePtrType cur;

   success = ((position >= 1) &&
                            (position <= ListLength()));

   if ( success ) {
      cur      = PtrTo(position);
      dataItem = cur->item;
   }

   return;
}



//*****************************************************************
// ListUpdates: Updates data from specified node.                 *
//                                                                *
// INPUT:  int           position - Position of node from which the *
//                                  data is to be retrieved.      *
//         instance      dataItem - The parameter into which the  *
//                                  desired item is retrieved     *
//                                                                *
// LOCAL:  instanceNodePtrType cur- pointer to the desired item   *
// OUTPUT: int   &       success  - Returns TRUE if the retrieve  *
//                                  was successful, FALSE         *
//                                  otherwise                     *
//*****************************************************************

void
instanceClass::ListUpdate(int position, instance dataItem,
                        int& success)
{
   instanceNodePtrType cur;

   success = ((position >= 1) &&
                            (position <= ListLength()));

   if ( success ) {
      cur      = PtrTo(position);
      cur->item = dataItem;
   }

   return;
}
```

```
//*****************************************************************
// ListInsert: Inserts new node into list.                        *
//                                                                *
// INPUT:  int          newPosition - the position where to insert *
//                                    the new node                *
//         instance     newItem     - The item to insert           *
//                                                                *
// LOCAL:  int          newLength   - list length after insertion  *
//         instanceNodePtrType newPtr- node to hold new value     *
//                              prev  - points to previous node for*
//                                    reassigning pointers        *
//                                                                *
// OUTPUT: int&  success            - TRUE if the insert was      *
//                                    successful, otherwise FALSE *
//                                                                *
//*****************************************************************


void
instanceClass::ListInsert(int newPosition, instance newItem,
                     int& success)
{
   int     newLength;
   instanceNodePtrType newPtr, prev;

   newLength = ListLength() + 1;

   success =((newPosition >= 1) &&
                     (newPosition <= newLength));

   if ( success ) {
      size = newLength;

      newPtr  = new instanceNode;
      success =(newPtr != NULL);

      if ( success ) {
         newPtr->item = newItem;

         if ( newPosition == 1 ) {
            newPtr->next = head;
            head         = newPtr;
         } else {
            prev         = PtrTo(newPosition - 1);
            newPtr->next = prev->next;
                prev->next   = newPtr;
         }
      }
   }

   return;
}
```

```
//*********************************************************************
// ListDelete: Delete node from the list.                             *
//                                                                    *
// INPUT:  int      position - position of node to be deleted.        *
// LOCAL:  instancNodePtrType  cur - saved pointer (to head or next)* *
//                             prev- pointer to the previous cell     *
// OUTPUT: int&  success  - TRUE if the delete was successful,        *
//                          FALSE otherwise.                          *
//                                                                    *
//*********************************************************************

void
instanceClass::ListDelete(int position, int& success)
{
   instanceNodePtrType cur, prev;

   success = ((position >= 1) &&
                            (position <= ListLength()));

   if ( success ) {
      size--;

      if ( position == 1 ) {
         cur  = head;
         head = head->next;
      } else {
         prev       = PtrTo(position - 1);
         cur        = prev->next;
         prev->next = cur->next;
      }

      cur->next = NULL;
      delete cur;
      cur       = NULL;
   }
}


//*********************************************************************
// instanceClass: assignment =                                        *
//                                                                    *
// INPUT: const   instanceClass& sourcelist - The source list to copy *
// LOCAL: instanceNodePtrType newPrev      - The new list head        *
//        instanceNodePtrType origCur      - local ptr                *
//                                                                    *
//*********************************************************************

void
instanceClass::operator=(const instanceClass& sourcelist)
{

   instanceNodePtrType newPrev, origCur;
   size = sourcelist.size;
   if ( sourcelist.head == NULL )
      head = NULL;
   else {
```

```
      head = new instanceNode;
      assert(head != NULL);
      head->item = sourcelist.head->item;

      newPrev = head;

      for ( origCur =  sourcelist.head->next;
            origCur != NULL;
            origCur =  origCur->next ) {
         newPrev->next = new instanceNode;
         assert(newPrev->next != NULL);
         newPrev       = newPrev->next;
         newPrev->item = origCur->item;
      }

      newPrev->next = NULL;
   }
}



//*****************************************************************
// displayInstanceList: Displays the list created                 *
// LOCAL:  int instancePosition - the position where to dispay node*
//         timeType timeTemp   - temp variable for time instance   *
//         int  timePosition   - counter for time position in array*
//         int    success      - true when node retrieved form list*
//         instance instanceTemp - the item to display             *
//*****************************************************************

void instanceClass::displayInstanceList()
{
      instance    instanceTemp;
      timeType    timeTemp;
      int instancePosition = 1;
      int timePosition     = 0;
      int success;

      while(instancePosition <= ListLength())
      {
            ListRetrieve(instancePosition, instanceTemp, success);

            while(timePosition < instanceTemp.appCounter)
            {
            timeTemp = instanceTemp.timeArray[timePosition];

            //1 added to timePosition because the array begins at zero

            cout<<" T"<<timePosition + 1<<' ';
            cout<<"sec=> "<<timeTemp.tv_sec<<", usec => ";
            cout<<timeTemp.tv_usec<<endl;

            timePosition++;
            }//while
```

```
            timePosition = 0;
            instancePosition++;
      }//while
}//end

//*************************************************************************
// instanceComplete: Determines if all time positions in array are      *
//                   filled                                             *
// RETURN: pathLength==number of applications found in current instance*
//         int    success  - true when node retrieved form list        *
//*************************************************************************

int instanceClass::instanceComplete(int pathLength, instance
currentInstance)

{
      return(pathLength == currentInstance.appCounter);
}
```

C. PATHTIMER MODULE

```
/*************************************************************************
// FILE: pathTimer.cpp                                                  *
//                                                                      *
// AUTHOR: Kendal Polk                                                  *
//                                                                      *
// LAST MODIFIED: 6 June 2000                                           *
//                                                                      *
// PURPOSE: This is the source file for operations necessary for the*
//          pathTimer Module                                            *
//                                                                      *
//                                                                      *
//*************************************************************************/

#include "pathTimer.h"

//*************************************************************************
// calcTotalPathTime: Based on current path instance calculates the  *
//                    total path time (TPT)                          *
//                                                                   *
// INPUT:  instance currentInstance                                  *
//         int lastAppPosition                                       *
//                                                                   *
// LOCAL:  timetype   firstAppTime                                   *
//                    lastAppTime                                    *
//                    returnApp                                      *
//                                                                   *
// RETURN: TPT (returnApp)                                           *
//                                                                   *
//*************************************************************************

timeType calcTotalPathTime(instance currentInstance)

{
      timeType firstAppTime, lastAppTime,returnApp;
```

```
      int lastAppPosition;


      lastAppPosition = currentInstance.appCounter - 1;


      firstAppTime = currentInstance.timeArray[0];

      lastAppTime  = currentInstance.timeArray[lastAppPosition];


      returnApp.tv_sec = (lastAppTime.tv_sec - firstAppTime.tv_sec);
        returnApp.tv_usec= (lastAppTime.tv_usec- firstAppTime.tv_usec);

        //correct for a negative fraction of second
        if(returnApp.tv_usec < 0){

           if(returnApp.tv_sec >= 0){
              returnApp.tv_sec -= 1;
              returnApp.tv_usec += 1000000;
           }
           else{
              returnApp.tv_usec *= -1;
           }//end if-else
        }
        else if(returnApp.tv_sec < 0){
           returnApp.tv_sec  += 1;
           returnApp.tv_usec = 1000000 - returnApp.tv_usec;
        }//end if

      return(returnApp);
}// end
```

D. EVALUATE AND ALERT MODULE

```
/*********************************************************************
// FILE: evalalert.cpp                                                *
//                                                                    *
// AUTHOR: Kendal Polk as a modification of Template from Data        *
//         Abstraction and Problem solving with C++ by Frank          *
//         Carrano, CH4,pp165.                                        *
//                                                                    *
// LAST MODIFIED: 6 June 2000                                         *
//                                                                    *
// PURPOSE: This is the source file for operations necessary for the*
//          evalalert Module                                          *
//                                                                    *
//                                                                    *
//*********************************************************************/

#include "evalalert.h"
#include "pathDBInstance.h"
```

```
//*********************************************************************
// goodInstance: Based on totalpathtime(TPT)  and path deadline, this*
//               function returns true if TPT <= deadline            *
//                                                                   *
// INPUT:  timetype   totalPathTime                                  *
//                    deadline                                       *
//                                                                   *
// RETURN: 1 for true and 0 for false                                *
//                                                                   *
//*********************************************************************

int goodInstance(timeType totalPathTime,timeType deadline)

{
      int good = 0;

      cout<<totalPathTime.tv_sec<<" TPT  "<<deadline.tv_sec<<" DL
        secs\n";
      cout<<totalPathTime.tv_usec<<" TPT  "<<deadline.tv_usec<<" DL
        secs\n";

      if(totalPathTime.tv_sec > deadline.tv_sec){
          good = 0;
        }
        else
          if((totalPathTime.tv_sec == deadline.tv_sec)&&
             (totalPathTime.tv_usec > deadline.tv_usec)){
                good=0;

        }//if-else


        return(good);
      }//end



//*********************************************************************
// evaluate : Based on false result from goodInstance rasies flag    *
//                                                                   *
// INPUT:  timetype   totalPathTime                                  *
//                    deadline                                       *
//*********************************************************************

void evaluate(timeType deadline, timeType totalPathTime)

{

    #ifdef TPMS_DEBUG
          cout<<" I am here in evalalert\n";
    #endif

    //lets see if this is a good instance
```

```
    if (!goodInstance(totalPathTime,deadline))
    {
      raiseFlag();
    }//end if
}//end
```

```
//*********************************************************************
// raiseFlag : This function emulates the identification of path not *
//             meeting its defined dealine. Actions derived from this*
//             function should be defined in later work.             *
//*********************************************************************

void raiseFlag()

{
      cout<<"\n Flag Raised \n";
      //Should be developed later;

}//end
```

E. CONTROL MODULE

```
/*********************************************************************
// FILE:tpms.cpp                                                      *
//                                                                    *
// AUTHOR: Kendal Polk                                                *
//                                                                    *
// LAST MODIFIED: 21 June 2000                                        *
//                                                                    *
// PURPOSE: This is the entry point for "wrapped" applications to     *
//         report their start and completion within their respective*
//         paths.  This file represents the control module in the     *
//         TPMS architecture.                                         *
//                                                                    *
//*********************************************************************/


#include "specDB.h"
#include <stddef.h>
#include "pathDB.h"
#include <sys/types.h>
#include<sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <errno.h>
#include <time.h>
#define BUFLEN 14
//#define TPMS_DEBUG //uncomment here to receive debug statements
```

```
char appStatus;


int main(int argc,char* argv[])
{

   SpecDBClass specDBList;

   pathDBClass pathDBList;

//make the spec Database

   specDBList.createSpecDB(argv[1]);

   specDBList.displayList();

//build the initial path Database

   pathDBList.createPathDB(specDBList);

   pathDBList.displayPathList();


//set up to receive input from "wrapped" applications

   struct timeval currentTime,
                  *currentTimePtr = &currentTime;

   struct timezone timeZone,
                   *timeZonePtr = &timeZone;


   int sockMain, addrLength, msgLength;

   struct sockaddr_in servAddr, clientAddr;

   char buf[BUFLEN], *leftPtr,*timePtr;


   if ((sockMain=socket(AF_INET,SOCK_DGRAM, 0)) <0)
      {perror("could not get sockect.\n");
      exit(1);
      }

   bzero((char*) &servAddr, sizeof(servAddr));

   servAddr.sin_family = AF_INET;
   servAddr.sin_addr.s_addr =htonl(INADDR_ANY);

   //this port is hardcoded so that wrapped applications know
   // to send their QoS output
   servAddr.sin_port=50001;

   if (bind(sockMain,(struct sockaddr *)&servAddr, sizeof(servAddr))<0)
```

```
      {perror("Can't get port.\n");
      exit(1);
      }

   addrLength = sizeof(servAddr);
   if (getsockname(sockMain,(struct sockaddr *)&servAddr, &addrLength))
        {perror(" getsockname failed.\n");
      exit(1);
      }

   printf("\nserver: port Number is %d\n", ntohs(servAddr.sin_port));


   //This infinite loop listens on the specified socket for reports
   // from the Clients Libraries reporting from wrapped applications

   for (;;){

      addrLength =sizeof(clientAddr);
      if (!(bzero (buf,BUFLEN))){
            printf("ouch \n");}
      if((msgLength =recvfrom(sockMain,buf, BUFLEN,0,
                  (struct sockaddr *)&clientAddr,
                  &addrLength))<0)
                  {perror("bad client socket\n");
                  exit(1);
                  }//perror

        #ifdef TPMS_DEBUG
          printf("\nserver: Client's IP address is: %s  ",
                        inet_ntoa(clientAddr.sin_addr));

          printf(" port: %d \n", ntohs(clientAddr.sin_port));
        #endif

      //determine the event character
      appStatus = buf[0];

      // determine the ID of the application
      applicationID appID = &buf[1];

      //get reporting time from host computer
      gettimeofday(currentTimePtr, timeZonePtr);

      pathDBList.addAppTime(appStatus, appID,currentTimePtr,specDBList);

      #ifdef TPMS_DEBUG
          pathDBList.displayPathList();
      #endif

   }//for

      return 0;
}//
```

LIST OF REFERENCES

[CARR96] Frank Carrano, *Data Abstraction and Problem Solving with C++*, Benjamin/Cummings Publishing Co., California,1995.

[CASE91] Fred Case, Christopher Hines, and Steven Satchwell, *Analysis of Air Operations During Desert Shield / Desert Storm,* U.S. Air Force Studies and Analyses Agency, 1991.

[CHAT98] Saurav Chatterjee, *Dynamic Application Structuring on Heterogeneous, Distributed Systems,* SRI International, Menlo Park, California, 1998.

[CINC97] Joint IT-21 Message CINCLANFLT/CINCPACFLT, *Information Technology for the 21st Century*, Pearl Harbor, HI, Mar 97.

[CLAR97] Raymond Clark, E. Douglas Jensen, Arkady Kanevsky, John Maurer, Paul Wallace, *An Adaptive, Distributed Airborne Tracking System*, The MITRE Corporation, Bedford Massachusetts, 1997.

[DEIT98] H.M. Deitel, P.J. Deitel, *C++ How to Program,* Prentice Hall, New Jersey, 1994.

[DESI98] Dr. Roberto Desimone, Andrew Preece, and Simon Hall, Improving Command Decision Making through the Integration of Joint Planning Aids into C2 Systems, *Proceedings of the 1998 Command and Control Research and Technology Symposium,* Naval Postgraduate School, Monterey, California, 1998.

[DESI98] *Desiderata: Resource and QoS Management for Dynamic, Scalable, Dependable, Real-Time Systems*, Department of Computer Science and Engineering, University of Texas at Arlington, 1998.

[DRAK99] Tim Drake, *Distributed Real Time Application Emulator*, Master's Thesis, NPS Monterey, CA, September 1999.

[EADS98] Teledyne Brown Engineering, EADSIM Version 7.0, August 1998.

[FAR 96] *Federal Acquisition Regulations*. Washington, DC: General Services Administration, 1996.

[FRAS99] Tim Fraser, Lee Badger, and Mark Feldman, "Hardening COTS Software with Generic Software Wrappers,"*IEEE Symposium on Security and Privacy*, 9-12 May, 1999, Oakland, California

[FREU98] R. F. Freund and others, Scheduling Resources in Multi-user, Heterogeneous, Computing Environments with SmartNet, *Proceedings Eighth Heterogeneous Computing Workshop,* IEEE Computer Society, Los Alamitos, California, March 1998.

[HENS99] Debra A. Hensgen and others, An Overview of MSHN: The Management System for Heterogeneous Networks, *Proceedings Eighth Heterogeneous Computing Workshop(HCW '99),* IEEE Computer Society, Los Alamitos, California, 1999.

[HUNT99] Galen Hunt, Doug Brubacher, "Detours: Binary Interception of Win32 Functions", Microsoft Research, Washington, February 1999.

[HUST00] Geoff Huston, "Quality of Service—Fact or Fiction?",Cisco Publications,March 2000.

[JAIN91] Raj Jain, *The Art of Computer Systems Performance Analysis*, John Wiley and Sons, Inc., New York, 1991.

[JONG99] Jong-Kook Kim and others, *Priorities, Deadlines, Versions, and Security in a Performance Measure Framework for Distributed Heterogensous Networks.* In preparation for submission (1999).

[KIDD96] Taylor Kidd, Debbie Hensgen, Richard Freund, and Lantz Moore, "SmartNet: A Scheduling Framework for Heterogeneous Computing," *ISPAN*, 1996.

[KIDD99] Taylor Kidd and others, *Compute Characteristics*, Technical Report in progress, Naval PostGraduate School, Monterey, California, 1999.

[OBER97] Tricia Oberndorf, *COTS and Open Systems—An Overview,* Carnegie Mellon University, 1997.

[PORT99] N. Wayne Porter, *Resource Usage for Adaptive C4I Models in a Heterogeneous Cosmputing Environment*, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1999.

[SCHN98] Matthew C. L. Schnaidt, *Design, Implementation, and Testing of MSHN's Application Resource Monitoring Library,* Master's Thesis, Naval Postgraduate School, Monterey, California, September 1997.

[SKID99] Shirley Kidd and Matthew Schnaidt, *Tutorial on Wrapping Calls,* Naval Postgraduate School, 1999.

[SLED98] Carol Sledge, David Carney, *Case Study: Evaluating COTS Products for DoD Information Systems,* CarnegieMellon, SEI, June 1998.

[SONA00] Sonali Bhide, Private Communication, sonalib@bobcat.ent.ohiou.edu, 7 March 2000.

[UTAR96] Laboratory for Parallel and Distributed Real-time Systems, *DeSiDeRaTa: Resource and QoS Management for Dynamic, Scalable, Dependable Real-Time Systems,* University of Texas at Arlington, 1996.

[VAN 85] R. van Renesse and A.S. Tanenbaum, "Distributed Operating Systems", *ACM Computing Surveys*, Vol. 17, No. 4, Dec. 1985, pp.419-470.

[VIRT98] *COTS*, Vitual Town Hall Questions & Answers, www.acq-ref.navy.mil/vth/cots.html,1998.

[ZAND99] Victor C. Zandy, Barton P. Miller, Miron Livny,*Process Hijacking* University of Wisconsin, 1999.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

No. Copies

1. Defense Technical Information Center 2
   8725 John J. Kingman Rd., Ste 0944
   Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library 2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, CA 93943-5101

3. Chairman, Code CS 1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943-5000

4. Dr. Cynthia E. Irvine 7
   Computer Science Department Code CS/Ic
   Naval Postgraduate School
   Monterey, CA 93943-5000

5. Timothy Levin 1
   Computer Science Department Code CS
   Naval Postgraduate School
   Monterey, CA 93943-5000

6. CPT Kendal Polk 2
   3903 Daleview Terrace
   Chattanooga, TN 37411

69 290NPG TH 2865
6/02 22527-200 NLB

DUDLEY KNOX LIBRARY
3 2768 00404374 5